(19) JAPANESE PATENT OFFICE

PATENT ABSTRACTS OF JAPAN

(11) Publication number: **11085534 A**

(43) Date of publication of application: **30 . 03 . 99**

(51) Int. Cl **G06F 9/45**

(21) Application number: **10184702**

(22) Date of filing: **30 . 06 . 98**

(30) Priority: **30 . 06 . 97 US 97 885008**

(71) Applicant: **SUN MICROSYST INC**

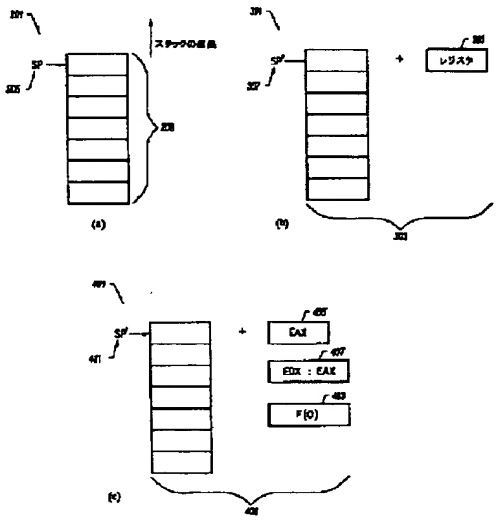
(72) Inventor: **GRIESEMER ROBERT**

(54) GENERATION AND PROVISION FOR INTERPRETER TO UTILIZE INTERPRETER STATE AND REGISTER CACHING

(57) Abstract:

PROBLEM TO BE SOLVED: To accelerate the execution speed of an interpreted computer program and further to provide an efficient interpreter concerning resources required for the interpreter.

SOLUTION: Concerning this method, the execution speed of a program to be interpreted is accelerated by using an operand stack. The most significant values of the operand stack are stored in registers 305, 405, 407 and 409 more than one. The state of the interpreter shows the data type of the most significant operand stack values stored in the registers more than one. Concerning a memory requested for the interpreter, the high-speed efficient interpreter can be generated.



(19)日本国特許庁（ＪＰ）　(12)公開特許公報（Ａ）　(11)特許出願公開番号

特開平11－85534

(43)公開日　平成11年(1999)３月30日

(51)Int.Cl.[6]	識別記号	ＦＩ	
Ｇ０６Ｆ　9/45		Ｇ０６Ｆ　9/44	３２０Ｃ

審査請求　未請求　請求項の数16　ＯＬ　外国語出願　（全 54 頁）

(21)出願番号　特願平10－184702

(22)出願日　平成10年(1998)６月30日

(31)優先権主張番号　０８／８８５００８

(32)優先日　1997年６月30日

(33)優先権主張国　米国（ＵＳ）

(71)出願人　595034134
サン・マイクロシステムズ・インコーポレイテッド
Ｓｕｎ　Ｍｉｃｒｏｓｙｓｔｅｍｓ，　Ｉｎｃ．
アメリカ合衆国　カリフォルニア州　94303　パロ　アルト　サン　アントニオ　ロード　901

(72)発明者　ロバート　グリーセマー
アメリカ合衆国，　カリフォルニア州，　メンロ　パーク，　オーク　レーン　960，　ナンバーエフ

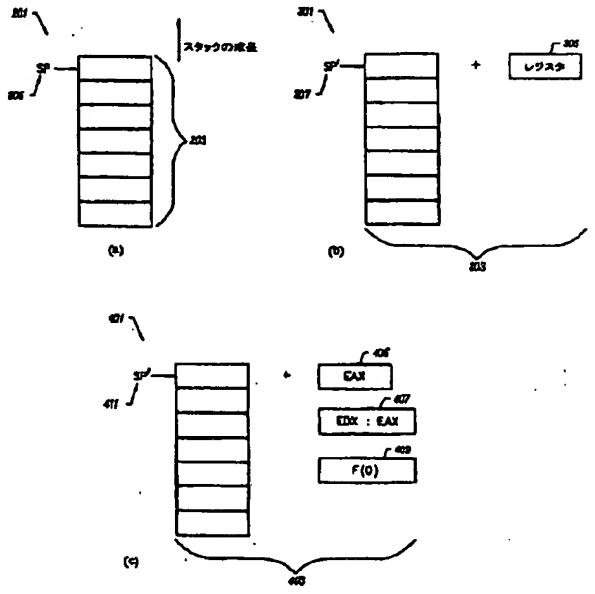
(74)代理人　弁理士　長谷川　芳樹　（外５名）

(54)【発明の名称】　インタプリタ状態及びレジスタキャッシングを利用するインタプリタの発生及び実現

(57)【要約】

【課題】　インタープリットされているコンピュータプログラムの実行速度を向上させる新しい技法に対する必要性がある。さらに、インタプリタが必要とする資源に関して、効率的なインタプリタを提供するための必要性がある。

【解決手段】　オペランドスタックを使って、インタープリットされるプログラムの実行速度を向上させるシステムと方法が提供される。オペランドスタックの最上位のための値は１つ以上のレジスタ（３０５、４０５、４０７、４０９）に格納される。インタプリタの状態は、１つ以上のレジスタに格納されるオペランドスタックの最上位のための値のデータタイプを示している。インタプリタに要求されているメモリに関して高速で効率的なインタプリタを生成することができる。



1

【特許請求の範囲】

【請求項１】　複数のレジスタを含むコンピュータシステムにおいて最上位を有するオペランドスタックを含むインタプリタをインプリメントする方法であって、
該複数のレジスタのうちの少なくとも１つのレジスタに該オペランドスタックの最上位のための値を格納するステップと、
該少なくとも１つのレジスタに格納されている該オペランドスタックの最上位のための該値のデータタイプを示すために、インタプリタの状態を利用するステップと、を備える方法。

【請求項２】　該データタイプは、整数、倍長整数、単精度浮動小数点数、および倍精度浮動小数点数からなる群から選択される、請求項１に記載の方法。

【請求項３】　該データタイプは、該オペランドスタックの最上位のための該値が該少なくとも１つのレジスタに現在のところ格納されていないことを示すボイドである、請求項１または２に記載の方法。

【請求項４】　該オペランドスタックの最上位を利用する命令は、該少なくとも１つのレジスタをアクセスする、請求項１～３のいずれか１つに記載の方法。

【請求項５】　該複数のレジスタは、異なるデータタイプの値を格納するために使用される少なくとも２つのレジスタを含む、請求項１～４のいずれかに記載の方法。

【請求項６】　該インタプリタの状態によって示されたデータタイプは、該オペランドスタックの最上位を格納する該複数のレジスタうちの少なくとも１つを指定する、請求項１～５に記載の方法。

【請求項７】　該インタプリタはＪａｖａ仮想マシンである、請求項１～６のいずれかに記載の方法。

【請求項８】　複数のレジスタを含むコンピュータシステムにおいて該少なくとも１つのレジスタにオペランドスタックの最上位のための値を格納するインタプリタを発生する方法であって、
該インタプリタによってインタープリットされるべき仮想マシン命令を選択するステップを備え、
該インタプリタの状態を選択するステップを備え、該インタプリタの状態は、該複数のレジスタの少なくとも１つのレジスタに格納される該オペランドスタックの最上位のための値のデータタイプを示し、
該選択された状態が期待された状態と異なる場合、該選択された仮想マシン命令に対する該期待された状態に該インタプリタを置くための、インタプリタのためのコンピュータコードを発生するステップを備え、
該インタプリタのためのコンピュータコードを生成し、該選択された仮想マシン命令を実行するステップを備える、方法。

【請求項９】　該選択された仮想関数のための該期待された状態は、仮想マシン命令を実行する前に該インタプリタの期待された状態を格納する、仮想マシン命令によってインデックス付けされる表をアクセスすることによって、獲得される、請求項８に記載の方法。

2

【請求項１０】　該表は、該仮想マシン命令の実行後の該インタプリタの現在の状態を格納する、請求項９に記載の方法。

【請求項１１】　該表は、該インタプリタのためのコンピュータコードを発生し該仮想マシン命令を実行する関数へのポインタを格納する、請求項９に記載の方法。

【請求項１２】　該選択された関数を実行する該インタプリタのためのコンピュータコードは、該インタプリタのためのコンピュータコードを発生し該仮想マシン関数を実行する関数へのポインタを格納する、仮想マシン命令によってインデックス付けされた表に指定された関数を呼ぶことによって発生される、請求項８～１１のいずれかに記載の方法。

【請求項１３】　次の仮想マシンコードをフェッチする該インタプリタのためのコンピュータコードを発生するステップを、更に備える請求項８～１２に記載の方法。

【請求項１４】　該選択された仮想マシン命令の実行の後のインタプリタの現在の状態のための次の仮想マシン命令を実行する該インタプリタにおけるロケーションへジャンプするための該インタプリタのためのコンピュータコードを発生するステップを、更に備える請求項８～１３のいずれかに記載の方法。

【請求項１５】　該選択された仮想マシン命令が選択された状態にとって非合法である場合、インタプリタにおける該ロケーションはエラーを処理する、請求項８～１４の何れかに記載の方法。

【請求項１６】　該インタプリタはＪａｖａ仮想マシンである、請求項８～１５に記載の方法。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】本発明は、ソフトウェアインタプリタのインプリメンテーション及び生成に関する。より具体的には、スタックベースの（stack-based）操作を最適化するためにレジスタを利用するＪａｖａ™仮想マシンのためのインタプリタを実現し、また生成することに関する。

【０００２】

【従来の技術】Ｊａｖａ™プログラム言語は、サンマイクロシステムズによって開発され、また小さなパーソナルコンピュータからスーパーコンピュータに至る幅広い範囲のコンピュータ上において実行されるに持ち運びに十分であるように設計された、オブジェクト指向の高級プログラミング言語である。Ｊａｖａ™（または他の言語）で記述されたコンピュータプログラムは、Ｊａｖａ™仮想マシンによって実行されるための仮想マシン命令にコンパイルされることができる。一般に、Ｊａｖａ™仮想マシンは、仮想マシン命令をデコードし、また実行するインタプリタである。

【０００３】Ｊａｖａ™仮想マシンのための仮想マシン命令はバイトコードであり、そのことは、その仮想マシン命令が１以上のバイトを含むことを意味する。そのバイトコードは、「クラスファイル」と呼ばれる特別のファイル形式に格納される。そのバイトコードに加えて、クラスファイルはシンボルテーブルおよび他の付属の情報を含んでいる。

【０００４】１つ以上のクラスファイルにおいてＪａｖａバイトコードとして具現化されているコンピュータプログラムは、プラットフォームに依存しない。コンピュータプログラムは、Ｊａｖａ仮想マシンのインプリメンテーションを走らすことができる任意のコンピュータ上において、変形されずに（unmodified）、実行されることができる。Ｊａｖａ仮想マシンは、Ｊａｖａ仮想マシンのためのコンピュータプログラムがプラットフォームに依存しないことを可能にしている主な因子である、「一般的な（generic）」コンピュータのソフトウェアエミュレータである。

【０００５】

【発明が解決しようとする課題】Ｊａｖａ仮想マシンは、一般にソフトウェアインタープリタとして実現される。伝統的なインタープリタは、実行中にひと時に一命令毎に、インタープリットされた（interpret、解釈された）プログラムの仮想マシン命令をデコードし、また実行する。一方、コンパイラは、実行中にデコードが行われないように、実行に先だってソースコードを固有のマシン命令にデコードする。伝統的なインタプリタは、各命令に出会うたび毎に繰り返しその命令が実行される前に命令をデコードするので、コンパイルされたプログラムの固有のマシン命令は固有のマシンまたはコンピュータシステム上においてデコードを必要とすることなく実行されることができるので、インタープリットされたプログラムの実行は、典型的には、コンパイルされたプログラムの実行よりかなり遅くなる。

【０００６】インタープリットされたプログラムをデコードして、実行するために、ソフトウェアインタープリタが実行されなければならないので、ソフトウェアインタープリタは、資源（例えば、メモリ）を消費して、したがって、その資源は、インタープリットされたプログラムにもはや利用可能でない。これは、固有のマシン命令を実行するコンパイルされたプログラムとは際立って対称的であって、それゆえ、コンパイルされたプログラムは、対象のコンピュータ上において直接に実行されることができ、したがって、インタープリットされたプログラムよりも多くの資源を一般的に自由に利用できる。

【０００７】そのため、インタープリットされているコンピュータプログラムの実行速度を向上させる新しい技法に対する必要性がある。さらに、インタプリタが必要とする資源に関して、効率的なインタプリタを提供するための必要性がある。

【０００８】

【課題を解決するための手段】一般に、本発明のいくつかの実施例は、インタープリタによって実行されるコンピュータプログラムの実行速度を増加させる革新的なシステムおよび方法を提供する。このインタプリタは、仮想マシン命令を実行するために利用されるオペランドスタックを含む。オペランドスタックの最上位のための値は、１つ以上のレジスタに格納されて、この値は、スタックベースの仮想マシン命令の実行速度が向上されることを可能にする。インタープリタの状態は、１つ以上のレジスタに格納されたオペランドスタックの最上位ための値のデータタイプを示すために利用される。この発明の場合、プログラムは、レジスタを用いて、より効率的な方法でインタープリットされることができる。加えて、インタープリタのサイズは、小さく保たれ、より多くの資源がインタープリットされたプログラムのために利用可能であることを可能にする。この発明のいくつかの実施例が以下に記載される。

【０００９】ある実施例においては、オペランドスタックを含むインタプリタをインプリメントするためのコンピュータでインプリメントされる方法が提供される。オペランドスタックの最上位のための値は、スタック上に格納される代わりに、少なくとも１つのレジスタに格納される。多くの伝統的なコンピュータは、異なったデータタイプを格納するためのレジスタを有する。そのため、スタックの最上位のための値は、そのデータタイプに適した１つ以上のレジスタに格納されて、インタープリタの状態は、１つ以上のレジスタに格納されるオペランドスタックの最上位のための値のデータタイプを示すために利用される。好ましい実施例においては、インタプリタはＪａｖａ仮想マシンであり、またインタプリタの状態は、整数、倍長整数、単精度浮動小数点数、および倍精度浮動小数点数を含むことができる。

【００１０】別の実施例においては、１つ以上のレジスタにオペランドスタックの最上位のための値を格納するインタプリタを生成するための、コンピュータでインプリメントされる方法が提供される。インタープリタの状態は、１つ以上のレジスタに格納されるオペランドスタックの最上位のための値のデータタイプを示している。インタプリタを生成するために、コンピュータは、可能な全ての仮想命令およびインタプリタの状態にわたって（through）ループすることができる。各繰り返しにおいて、仮想マシン命令およびインタプリタの状態が選択されることができる。選択された状態が、選択された仮想マシン命令の実行に先立って期待されるインタプリタの状態と違っているならば、期待された状態にインタプリタを置くようにインタプリタのためのコンピュータコードが生成される。ひとたび、選択された仮想マシン命令の実行に先立って期待される状態にそのインタプリタがあることがわかると、選択された仮想マシン命令を実

行するようにインタプリタのためのコンピュータコードが生成される。選択された仮想マシン命令に対する期待された状態は、仮想マシン命令を実行する前のインタプリタの期待される状態と、仮想マシン命令を実行した後の現在のインタプリタの状態とを格納し仮想マシン命令によってインデックス付けされたテーブルをアクセスすることによって獲得されることができる。さらに、選択された仮想マシン命令を実行するインタプリタのためのコンピュータコードは、テーブルにおいて特定された関数を呼ぶことによって生成されることができる。

【００１１】他の実施例においては、コンピュータで可読な媒体によって格納され、仮想マシン命令のインタプリタのためのデータ構造が提供される。このデータ構造は、仮想マシン命令によってインデックス付けされ、且つ多重のフィールドを有するテーブルである。テーブルのある１つのフィールドには、仮想マシン命令の実行前のインタプリタの期待される状態が格納される。テーブルの別のフィールドには、仮想マシン命令の実行後のインタプリタの現在の状態が格納される。さらに、テーブルのフィールドは、仮想マシン命令を実行するためにインタプリタのためのコンピュータコードを生成する関数へのポインタを格納するために利用されることができる。好ましい実施例においては、インタプリタの状態は、１つ以上のレジスタに格納されているインタプリタのオペランドスタックの最上位のための値のデータタイプを示す。

【００１２】他の実施例においては、仮想マシン命令のインタプリタのためのコンピュータで可読な媒体によって格納されたデータ構造が提供される。データ構造は、仮想マシン命令によってインデックス付けされ、且つ多重のフィールドを有するテーブルであって、各フィールドは、インタプリタの状態に関連付けられ、またインデックス付けされた仮想マシン命令を実行するインタプリタ内のロケーションへのポインタを格納する。インタプリタの状態は、１つ以上のレジスタに格納されたインタプリタのオペランドスタックの最上位のための値のデータタイプを示すことができる。好ましい実施例においては、インタプリタの状態は、整数、倍長整数、単精度浮動小数点数、および倍精度浮動小数点数である。

【００１３】本発明の他の特徴や利点は、添付図面に関連して以下の詳細な説明を見れば容易に明らかになる。

【００１４】

【発明の実施の形態】

定義

マシン命令　－　オペレーションコード（オペコード）と随意に（optionally）１つ以上のオペランドによって特定される操作を実行することをコンピュータに指示する命令

仮想マシン命令　－　ソフトウェアでエミュレートされる（software emulated）マイクロプロセッサまたはコンピュータのアーキテクチャのためのマシン命令（仮想コードとも呼ばれる）。

固有の（Native）マシン命令　－　特定のマイクロプロセッサまたはコンピュータのアーキテクチャのために設計されたマシン命令（ネイティブコードとも呼ばれる）。　クラス　－　クラスの各オブジェクトが含むデータおよびメソッドを定義するオブジェクト指向のデータタイプ。

関数　－　ソフトウェアルーチン（サブルーチン、手続き、メンバ関数、およびメソッドとも呼ばれる）。

オペランドスタック　－　実行中にマシン命令によって使用するためにオペランドを格納するために利用するスタック。

バイトコードポインタ（ＢＣＰ）　－　実行されている現在のＪａｖａ仮想マシン命令（例えば、バイトコード）を指すポインタ。

プログラムカウンタ（ＰＣ）　－　実行されているインタプリタの（典型的には、固有の）マシン命令を指すポインタ。

インタプリタ　－　典型的にはコンピュータプログラムにおける各命令を翻訳し、次に実行するソフトウェアまたはハードウェア内のプログラム。

インタプリタ生成器　－　インタプリタを生成するソフトウェアまたはハードウェアにおけるプログラム。

【００１５】概要

以下に続く記述において、本発明が、Ｊａｖａ仮想マシン命令（バイトコード）を実行するためのＪａｖａ仮想マシンをインプリメントする（implement、実現する）好ましい実施例を参照して記述される。特に、ＩＢＭパーソナルコンピュータ（Ｉｎｔｅｌ　ｘ８６マイクロプロセッサアーキテクチャ）の固有のマシン命令を含んで例示が記述される。しかしながら、この発明は、特別の言語、コンピュータアーキテクチャ、または特定のインプリメンテーション（implementation、実現）に限られるものではない。したがって、以下に続く実施例の記述は、例示のためであって、限定のためではない。

【００１６】図１は、本発明の実施例のソフトウェアを実行するために使用されることができるコンピュータシステムの例を図示している。図１は、ディスプレイ３、スクリーン５、キャビネット７、キーボード９、マウス１１を含むコンピュータシステム１を示している。マウス１１は、グラフィカルユーザインターフェースを用いて対話する（interact）ための１つ以上のボタンを有することができる。キャビネット７は、本発明をインプリメントするコンピュータコードと、本発明とともに使用するデータ、などを組み込むソフトウェアプログラムを格納し、また検索するために使用することができるＣＤ－ＲＯＭドライブ１３、システムメモリ、およびハードドライブ（図２を見よ）を収納する。コンピュータで可読な例示的な記憶媒体としてＣＤ－ＲＯＭ１５が示され

ているけれども、フロッピーディスク、テープ、フラッシュメモリ、およびハードドライブを含むコンピュータで可読な他の記憶媒体を使用することもできる。さらに、（例えば、インターネットを含むネットワークにおける）キャリア波（carrier wave）で具現化されたデータ信号が、コンピュータで可読な記憶媒体であることもできる。

【００１７】図２は、本発明の実施例のソフトウェアを実行するために使用されるコンピュータシステム１のシステムブロック図を示す。図１にあるように、コンピュータシステム１は、モニタ３およびキーボード９、並びにマウス１１を含む。コンピュータシステム１は、更に、中央プロセッサ５１、システムメモリ５３、固定記憶装置５５（例えば、ハードドライブ）、リムーバブル（removable、着脱可能な）記憶装置５７（例えば、ＣＤ－ＲＯＭドライブ）、ディスプレイアダプタ５９、サウンドカード６１、スピーカ６３、およびネットワークインターフェース６５といった、サブシステムを含む。本発明とともに使用するために適した他のコンピュータシステムは、付加的な、または、少数のサブシステムを含むこともある。例えば、別のコンピュータシステムは、１つより多いプロセッサ５１（例えば、マルチプロセッサシステム）、キャッシュメモリを含むこともできるでしょう。

【００１８】コンピュータシステム１のシステムバスアーキテクチャは、矢印６７で表わされる。しかしながら、これらの矢印は、サブシステムをリンクするために役立つ相互接続のスキームの例示である。例えば、中央プロセッサをシステムメモリおよびディスプレイアダプタに接続するためにローカルバスを利用することもできるでしょう。図２に示されたコンピュータシステム１は、本発明と共に使用するために適したコンピュータシステムの単なる例である。サブシステムの異なった構成を有する他のコンピュータアーキテクチャを利用することができる。

【００１９】典型的には、Ｊａｖａプログラミング言語で記述されたコンピュータプログラムは、Ｊａｖａ仮想マシン命令またはバイトコードにコンパイルされ、これらは、その後に、Ｊａｖａ仮想マシンによって実行される。そのバイトコードは、インタープリテーションのためにＪａｖａ仮想マシンへ入力されるクラスファイルに格納される。図３は、インタープリタである、Ｊａｖａ仮想マシンによる実行を通して、単純な一片のＪａｖａソースコードの進行（progression）を示している。

【００２０】Ｊａｖａソースコード１０１は、Ｊａｖａで記述された古典的なＨｅｌｌｏＷｏｒｌｄプログラムを含んでいる。ソースコードは、その後に、ソースコードをバイトコードにコンパイルするバイトコードコンパイラ１０３に入力される。バイトコードは、ソフトウェアがエミュレートする（software emulated）コンピュータによって実行されるので、仮想マシン命令である。典型的には、仮想マシン命令はジェネリック（generic）である（すなわち、特定のマイクロプロセッサまたはコンピュータのアーキテクチャのために設計されたものではない）が、しかし、このことは要求されていない。バイトコードコンパイラは、Ｊａｖａプログラムのためのバイトコードを含むＪａｖａクラスファイル１０５を出力する。

【００２１】Ｊａｖａクラスファイルは、Ｊａｖａ仮想マシン１０７へ入力される。Ｊａｖａ仮想マシンは、Ｊａｖａクラスファイル内のバイトコードをデコードして、実行するインタプリタである。このＪａｖａ仮想マシンは、インタプリタであるが、しかしこれはマイクロプロセッサまたはコンピュータのアーキテクチャ（例えば、ハードウェアには存在しないかもしれないマイクロプロセッサまたはコンピュータアーキテクチャ）をソフトウェアでエミュレートするので、通例、仮想マシンとして参照される。

【００２２】インタプリタは、以下のプロセスを繰り返し実行することによってバイトコードプログラムを実行することができる。

実行　－　現在のバイトコードの操作を実行する。

進行　－　バイトコードポインタを次のバイトコードに進める。

ディスパッチ　－　バイトコードポインタにおけるバイトコードをフェッチして、そのバイトコードを実行すること（implementation）（すなわち、実行ステップ）へジャンプする。

実行ステップは、特定のバイトコードの操作を実行する（implement）。進行ステップは、バイトコードポインタが次のバイトコードを指すように、そのポインタをインクリメントする。最後に、ディスパッチステップは、現在のバイトコードポインタのところの（at）バイトコードをフェッチして、そのバイトコードを実行する固有のマシンコードの一片へジャンプする。バイトコードのための、実行－進行－ディスパッチシーケンスの実行は、一般に「インタープリテーションサイクル（implementation cycle）」と呼ばれる。

【００２３】好ましい実施例においては、インタプリタは前述のインタープリテーションサイクルを利用するけれども、本発明との関連において他の多くのインタープリテーションサイクルを利用できる。例えば、インタプリタは、ディスパッチ－実行－進行のインタプリテーションサイクルを実行することができ、または、各サイクルにおいてより多くの若しくはより少ないステップがあってもよい。したがって、この発明は、ここで記載された実施例に限られない。

【００２４】Ｊａｖａ仮想マシンの実現および生成

Ｊａｖａ仮想マシンのための仮想マシン命令は、スタックベースの命令を含む。このため、仮想マシンのオペラ

ンドうちの１つ以上は、オペランドスタックに格納されることがある。オペランドスタックを説明する前に、一般的なスタックについて議論することが利益があるかもしれない。

【００２５】図４（ａ）は、データ２０３を格納するスタック２０１を示している。このスタックを用いて、データ値をスタックの最上位（トップ）に「プッシュ」することができる。代わりに、データ値を「ポップ」してスタックの最上位から離す（off）ことができる。概念的には、スタックの最上位だけがアクセスされることができ、それで、そのスタックは、最も最近にスタックにプッシュされたデータが、ポップされスタックから離れるべき最初のデータであることを意味するファーストイン・ファーストアウト（ＦＩＦＯ）のデータ構造である、として知られている。スタックポインタ（ＳＰ）２０５は、スタック２０１の最上位を差し指す。このため、ＳＰは、データ値がスタック２０１上にプッシュされるとき、そしてポップされスタックから離されるとき変化する。簡単のために、ここに記載されたスタックは、メモリ内で上方に成長するように示され、また記載されているが、しかしながら、これは単なるグラフィカルな表現であって、当業者は、スタックが他の多くの方法（例えば、メモリの下の方へ成長する方法）で示され、且つ／または実現されることができることを容易に認識する。

【００２６】Ｊａｖａ仮想マシンのための仮想マシン命令は、図４（ａ）に示されたスタックに類似したオペランドスタックを実現することを求める。より具体的には、Ｊａｖａ仮想マシンにおけるオペランドスタックは、実行中にバイトコードによる使用のためにオペランドを格納するように利用される。以下は、どのようにオペランドスタックがＪａｖａ仮想マシンにおいて利用されるかを示すために助けとなり得る一例である。

【００２７】Ｊａｖａソースコードが、ステイトメントＸ：＝Ａ＋Ｂを含むとし、ここで、Ｘ、Ａ、Ｂは整数変数である。このステイトメントは、以下のバイトコードにコンパイルされる。

１．　ＩＬＯＡＤ　Ａ
２．　ＩＬＯＡＤ　Ｂ
３．　ＩＡＤＤ
４．　ＩＳＴＯＲＥ　Ｘ

各バイトコードに先行する「Ｉ」は、このバイトコードによって操作されたデータタイプが整数であることを示している。倍長（long）整数には「Ｌ」、単精度浮動少数点数（single-precision floating point）には「Ｆ」、倍精度浮動小数点数（double-precision floating point）には「Ｄ」によって命名される他のデータタイプのための対応するバイトコードが存在する。Ｊａｖａ仮想マシンにおいては、整数データタイプは３２ビットであり、倍長整数データタイプは６４ビットであり、単精度浮動小数点数データタイプは３２ビットであり、倍精度浮動小数点数データタイプは６４ビットである。以下に記述されるように、仮想マシンにおけるこれらのデータタイプのサイズは、仮想マシンをインプリメントするコンピュータシステムにおけるサイズと同じではない場合もある（例えば、ＩＢＭパーソナルコンピュータにおける通常の整数データタイプは、１６ビットであり、６４ビットの倍長整数のための直接のサポートはない）、そこで、Ｊａｖａ仮想マシン命令と、仮想マシンがインプリメントされているコンピュータに指図する固有のマシン命令とを区別することが重要である。

【００２８】最初のＩＬＯＡＤバイトコードは、変数Ａの値をオペランドスタック上にロードする。同様に、２番目のＩＬＯＡＤバイトコードは、変数Ｂの値をオペランドスタック上にロードする。バイトコードＩＡＤＤは、２つのデータ値をポップしてオペランドスタックから離して、この２つの値を加えて、その和をオペランドスタック上にプッシュする。予測されるように、ＩＳＴＯＲＥバイトコードは、データ値、この例では和の値、をポップしてオペランドスタックから離し、それを変数Ｘに格納する。この単純な例示は、Ｊａｖａバイトコードがどのようにオペランドスタックを使うかを概念的に示している。

【００２９】本発明のいくつかの実施例は、データ値がオペランドスタックにプッシュされるときにしばしば、引き続いて（オペランドスタックの最上位を修正しない介在（intervening）バイトコードを持ち、または持たずに）その値がポップされオペランドスタックから離れるという事実を利点に持つ。典型的には、オペランドスタックはメモリにインプリメントされるが、本発明のいくつかの実施例の場合には、オペランドスタックの最上位に関する値は、１つ以上のレジスタに格納される。レジスタはメモリより速いアクセス時間を有するので、オペランドスタックの最上位のためのアクセス時間は縮小され、インタープリット（解釈）されたプログラムのより高速なインタープリテーションに帰着する。

【００３０】図４（ｂ）は、データ３０３を格納するオペランドスタック３０１を示している。データ３０３は、レジスタ３０５に格納されたオペランドスタックの最上位のためのデータ値と、メモリ上のデータ値との両方を含む。スタックポインタ’（ＳＰ’）３０７は、オペランドスタック３０１の最上位の概念的には直後のデータタイプを差し指す。この議論は、単一のオペランドスタックに焦点を当てているることに注目すべきである。しかしながら、本発明に従ってインプリメントされることができる（例えば、異なるスレッドおよびメソッドのための）単一のコンピュータシステムに複数のオペランドスタックがあることがある。

【００３１】伝統的なコンピュータシステムでは、典型的には、あるレジスタは特定のデータタイプを格納する

ためにより適合している状態で、多くの異なったレジスタを有する。例えば、コンピュータは３２ビット幅のレジスタ、および６４ビット幅のレジスタを有することができる。仮想的なコンピュータ上の整数が３２ビット値であり、一方、倍長整数が６４ビット値であるならば、そのデータ値のサイズを満たすレジスタにそのデータを格納したことがより効率的である。さらに、コンピュータシステムは、単精度浮動小数点数または倍精度浮動小数点数のような特定のデータタイプを格納するように設計されたレジスタを有することがある。

【００３２】より具体的には、ＩＢＭパーソナルコンピュータ（Ｉｎｔｅｌ　ｘ８６アーキテクチャ）は、多くの異なったタイプのレジスタを含む。３２ビットレジスタ（例えば、ＥＡＸ）、および浮動小数点数レジスタ（例えば、Ｆ（０））がある。加えて、Ｉｎｔｅｌ　ｘ８６マイクロプロセッサのデータタイプのいくつかは、Ｊａｖａ仮想マシンにおける対応物とは異なったサイズを有する。例えば、通常の整数データタイプは、Ｉｎｔｅｌ　８０３８６マイクロプロセッサにおいて、１６ビット幅であり、また倍長整数データタイプは、３２ビット幅である。これらのデータタイプは、Ｊａｖａ仮想マシンの対応物の半分のサイズである。そのため、Ｊａｖａ仮想マシンがｘ８６マシン上にインプリメントされている好ましい実施例においては、Ｊａｖａ仮想マシンにおける整数は、ｘ８６マシンにおける倍長整数と同じサイズ（すなわち、両方とも３２ビット値）である。

【００３３】以前に述べたように、命令が仮想マシンの命令、または、固有のマシン命令のどちらかであることを心にとどめておくことは重要である。Ｊａｖａ仮想マシン命令は、アセンブリ言語と似ていて、幸いなことに、簡単に識別可能な違いがある。Ｊａｖａ仮想マシンのために記述されたバイトコードにおいては、そのバイトコードが関係する（pertain）データタイプがその命令に先行する（例えば、ＩＳＴＯＲＥにおいて、「Ｉ」が整数を表わす）。これはデータタイプが命令に引き続く（例えば、ＰＯＰＬにおいて「Ｌ」は倍長整数を表わす）ｘ８６マイクロプロセッサのためのアセンブリコード（または、固有のマシン命令）とは際立って対照的である。このことは、本発明のすべての実施例に対して真実ではなさそうだけれども、この区別が、ここで記述された好ましい実施例の読者の理解を助けることが望まれる。

【００３４】図４（ｂ）に手短に戻ると、たった１つのレジスタのみが示され、またオペランドスタック３０１が異なったデータタイプを格納するために利用されることができるので、１つの問題があり、そのため、オペランドスタックの最上位のための値を格納するために利用可能な異なったレジスタを有することが望ましい。図４（ｃ）は、ｘ８６マイクロプロセッサ上にインプリメントされたＪａｖａ仮想マシンのためのオペランドスタック４０１を示している。オペランドスタック４０１は、レジスタ４０５、４０７、または４０９の１つに格納されたオペランドスタックの最上位のための値と、メモリ内のデータ値との両方を含むデータ４０３を格納する。好ましい実施例においては、レジスタ４０５は、オペランドスタックの最上位のための仮想マシン整数を格納するための３２ビットレジスタ（ＥＡＸ）である。レジスタ４０７は、オペランドスタックの最上位のための仮想マシン倍長整数を格納するための２つの３２ビットレジスタ（ＥＤＸ、ＥＡＸ）の組合わせである。レジスタ４０９は、単精度浮動小数点数と倍精度浮動小数点数との両方を格納するための６４ビット浮動小数点数レジスタ（Ｆ（０））である。特定のレジスタの指定（designation）は、本発明をより良く図示するように設けられ、しかしながら、本発明は、いかなる特定のレジスタおよびコンピュータアーキテクチャに限定されないことを理解すべきである。

【００３５】スタックポインタ’（ＳＰ’）４１１は、概念的にはオペランドスタック４０１の最上位の直後を差し指している。いま、オペランドスタックの最上位のための値を格納していることができる１つを越えるレジスタがあるとすると、どのレジスタ（register）またはレジスタ達（registers）がこの値を格納したかを知ることが望ましい。１つの技法は、オペランドスタックの最上位上の値のデータタイプを示す値をメモリ内にまたはレジスタ内に格納することであろう。このことによって、この値は、オペランドスタックの最上位を格納する正しいレジスタまたはレジスタ達を決定するためにアクセスされ得るであろう。

【００３６】この技法は作用する場合もあるけれども、その技法を不満足にするかもしれないいくつかの重要な欠点がある。例えば、オペランドスタックの最上位のデータタイプを特別に決定することは、オペランドスタックの最上位を格納するためにレジスタを利用するというパフォーマンスの向上を相殺することもある。

【００３７】本発明のいくつかの実施例の場合、インタプリタは、複数の状態において動作する。各状態は、１つ以上のレジスタに格納されたオペランドスタックの最上位の値のデータタイプを表わしている。その状態は、時間における任意の点においてインタプリタの本来備わっている（inherently）特質であり、このため、オペランドスタックの最上位のデータタイプを決定することは必要とされない。

【００３８】好ましい実施例においては、インタプリタは以下のような５つの異なった状態の１つにあり得る。

ＩＴＯＳ　－　オペランドスタックの最上位（ＴＯＳ）として（for）整数がレジスタ（register(s)）に格納される

ＬＴＯＳ　－　ＴＯＳとして倍長整数がレジスタ（register(s)）に格納される

ＦＴＯＳ － ＴＯＳとして単精度浮動小数点数がレジスタ（register(s)）に格納される
ＤＴＯＳ － ＴＯＳとして倍精度浮動小数点数がレジスタ（register(s)）に格納される
ＶＴＯＳ － ＴＯＳが現在のところレジスタ（register(s)）に格納されていないことを示すｖｏｉｄ ＴＯＳ
上記のように、ＶＴＯＳ状態は他の状態とは異なり、なぜならオペランドスタックの最上位が現在のところどのレジスタにも格納されていないことを示すからである。データ値がオペランドスタックにプッシュされ、ポップされ離されるとき、インタプリタはＶＴＯＳ状態と他の状態の１つとの間を交互に変動することが明らかであるべきである。

【００３９】インタプリタの異なった状態を管理することを手助けする（assist）ために、図５に示されるテンプレート表（データ構造）５０１が、本発明のいくつかの実施例において利用される。テンプレート表５０１は、バイトコード５０３によってインデックス付けされ、フィールド５０５、５０７、５０９を含む表である。テンプレート表５０１は、２００を越える（over）レコード（例えば、各バイトコードのために１つ）を有することができるけれども、本発明を最も良く表示すると考えられる部分集合のみが示されている。

【００４０】仮想マシン命令（または、バイトコード）が、テンプレート表５０１をインデックス付けするために利用される。フィールド５０５は、仮想マシン命令５０３が実行される前の、期待されているインタプリタの状態を格納する。例えば、ＩＳＴＯＲＥバイトコード（すなわち、オペランドスタックの最上位上に整数を格納する）が実行される前に、１つ以上のレジスタにオペランドスタックの最上位のための整数があることを示しているＩＴＯＳ状態にインタプリタがある、ことが期待されている。インタプリタが期待した状態に実行中にいないならば、それは必ずしもエラーを示すのではないが、しかし、以下に詳しく記述されるように、本発明の好ましい実施例は、バイトコード列において多くのエラーを検出できる。

【００４１】フィールド５０７は、仮想マシン命令５０３を実行するためにインタプリタのためのコンピュータコード（または、「テンプレート」、そしてこのため「テンプレート表」）を生成する関数へのポインタを格納する。好ましい実施例においては、関数の名前はバイトコードの名前と同一であり、その関数は、Ｃ++プログラミング言語で書かれている。

【００４２】最後に、フィールド５０９は、仮想マシン命令５０３を実行した後のインタプリタの現在の状態を格納する。例えば、ＩＳＴＯＲＥバイトコードが実行された後は、１つ以上のレジスタに格納された整数はポップされオペランドスタックから離されるので、インタプリタの現在の状態はＶＴＯＳである。

【００４３】好ましい実施例においては、フィールド５０５は、仮想マシン命令が実行される前の、期待されているインタプリタの状態を格納して、フィールド５０９は、仮想マシン命令が実行された後の、インタプリタの現在の状態を格納する。しかしながら、他の実施例においては、フィールド５０５は、フィールド５０７に明示されているテンプレート関数が実行される前の、期待されているインタプリタの状態を格納して、フィールド５０９は、このテンプレート関数が実行された後の、インタプリタの現在の状態を格納する。言い換えると、インタプリタの状態は、仮想マシン命令の代わりにテンプレート関数に基づくことができる。

【００４４】テンプレート表について説明してきたが、しかし、インタプリタの生成中に、他の表（「ディスパッチ表」）が、テンプレート表と結合して利用される。図６は、仮想マシン命令６０３によってインデックス付けされ、フィールド６０５、６０７、６０９、６１１、および６１３を含むディスパッチ表６０１の並びを示す。フィールド６０５は、ＩＴＯＳ状態のためのインデックス付けされた仮想マシン命令を実行するインタプリタにおけるロケーション（位置）またはアドレスへのポインタを格納する。同様に、フィールド６０７、６０９、６１１、および６１３は、それぞれ、ＬＴＯＳ状態、ＦＴＯＳ状態、ＤＴＯＳ状態、およびＶＴＯＳ状態のためのインデックス付けされた仮想マシン命令を実行するインタプリタ内のロケーションまたはアドレスへのポインタを格納する。このため、各フィールドは、インタプリタの状態に関連付けられている。フィールドの値は、生成されたインタプリタ内部へのポインタであるから、表示されていない。

【００４５】インタプリタは、典型的にはソフトウェアプログラム自身であることを思い出すと、ディスパッチ表は、インタプリタプログラムのコンピュータコード内の異なるロケーションへのジャンプ表である。言い換えると、実行されるべき次のバイトコードが、一旦、フェッチされると、インタプリタは、次のバイトコード（インデックスとして利用された）と、ジャンプのロケーションのためのフィールド６０５、６０７、６０９、６１１、および６１３のうちの１つを明示するインタプリタの現在の状態とによって明示されたディスパッチ表において指定されたロケーションへジャンプする。このように、インタプリタのプログラムカウンタは、ディスパッチ表に明示されたアドレスへセットされる。好ましい実施例においては、ディスパッチ表６０１は、各インタプリタの状態ために１つの、単一次元の表の５つとして実現されている。

【００４６】テンプレート表およびディスパッチ表は、１つの表（または、それについては２つを越える表）として実現されることができるのは明らかである。しかしながら、好ましい実施例においては、テンプレート表お

よびディスパッチ表は、別個の表であり、というのは、テンプレート表は、インタプリタの生成中に単独で利用される場合があり、したがって、インタープリタが生成された後に処分されることがある。ディスパッチ表は、インタープリタ生成中に生成され、または、値が埋められて、そして、インタプリタの実行中に有効に利用される。これらにもかかわらず、これらの表における情報は、当業者に知られている多くのデータ構造において多くの方法で実現されることができる。

【００４７】さて、テンプレート表およびディスパッチ表を説明してきたので、どのようにインタプリタが生成されるかを記述することは適切であるかもしれない。図７は、インタプリタを生成するプロセスを示している。一般的に、そのプロセスは、すべての仮想マシン命令およびインタプリタの状態の組み合わせにわたって循環することによって、インタプリタを生成する。これは、入れ子にされた（nested）ループ、単一ループ、または他の制御構造を用いて実現されることができる。好適な実施の形態においては、入れ子にされたループが利用される。

【００４８】ステップ７０１では、コンピュータシステムは、（例えば、仮想マシン命令にわたるループを通して１回繰り返すことによって）仮想マシン命令を選択する。システムは、インタプリタの状態をステップ７０３において選択する。一旦、仮想マシン命令およびインタプリタの状態が選択されると、図７における残りのプロセスは、インタプリタが、選択された状態に現在のところいるとき、選択された仮想マシン命令を扱う、インタプリタのためのコンピュータコードを生成する。図面は、例示の目的のために本発明の実施例のための流れ図を示しているけれど、これはステップの特定の順序または組み合わせを意味するものではない。一般的に、発明の範囲をはずれることなく、ステップは、順序を変えられ、組み合わされ、または削除されることができる。

【００４９】ステップ７０５においては、システムは、選択された仮想マシン命令およびインタプリタの状態が合法（legal）であるかを決定する。ある実施例においては、これは、テンプレート表（図５を見よ）を利用して、選択されたバイトコードに対するインタプリタの期待された状態を決定することによって達成される。期待された状態が選択された状態と同じならば、そのとき、選択された仮想マシン命令および状態の組み合わせは合法である。

【００５０】期待された状態が、選択された状態と異なるとしても、これは、この組み合わせが非合法（illegal）であることを必ずしも意味しない。代わりに、システムは、選択された状態から期待された状態への（オペランドスタックを破損する（corrupt）ことがないことを意味する）合法的な方法があるかどうかを決定する。例えば、期待された状態がＩＴＯＳであり、また選択された状態がＶＴＯＳであれば、インタプリタは、メモリに格納されているオペランドスタック内の最上位データ値を１つ以上のレジスタ内に移動する（例えば、ＳＰ’で指し示されたデータ値をレジスタに格納して、その後、ＳＰ’をデクレメントする）ことによって、ＩＴＯＳ状態に置かれることができる。他の例として、期待された状態がＩＴＯＳであり、また選択された状態がＤＴＯＳであるならば、オペランドスタックの最上位が現在のところ倍精度浮動小数点数であるので、インタプリタをＩＴＯＳ状態に置くための合法的な方法がない。

【００５１】一般的に、ＶＴＯＳ状態から任意の他の状態に移り、または、任意の他の状態からＶＴＯＳへ移ることは合法である。その理由は、インタプリタ状態のこのような遷移（shift）は、典型的には、メモリから１つ以上のレジスタへ、または、その逆にデータ値を移動することを含むからである。

【００５２】選択された仮想マシン命令およびインタプリタ状態が合法であるなら、システムはステップ７０７においてプロローグのコンピュータコードを生成することができる。プロローグのコンピュータコードは、選択された仮想マシン命令を実行する前に、都合よく生成される任意のコードである。一般的に、プロローグは、選択された仮想マシン命令および選択されたインタプリタ状態に依存する場合がある。例えば、（選択された仮想マシン命令のための）インタプリタの期待された状態が、選択されたインタプリタ状態と異なるならば、プロローグは、インタプリタを期待された状態に置くためのコンピュータコードを含むことができる。インタプリタの期待された状態および選択された状態が同じであれば、プロローグのコンピュータコードを生成する必要はない場合もある。

【００５３】ステップ７０９において、インタプリタが選択された仮想マシン命令を実行するようにコンピュータコードを生成するために、システムは、選択された仮想マシン命令のためのテンプレート関数を呼び出す。好ましい実施例においては、図５で示されたテンプレート表を選択された仮想マシン命令を用いてインデックス付けすることによってテンプレート関数が呼び出される。次に、そのテンプレート関数（または、その関数のためのアドレス）へのポインタを格納するフィールド、フィールド５０７がアクセスされ、そのテンプレート関数が呼び出される。

【００５４】テンプレート関数は、選択された仮想マシン命令を実行するためのコンピュータコードを生成する。テンプレート関数の例を少し説明することは役に立つであろう。前に述べたように、好ましい実施例においては、テンプレート関数は、ｘ８６マイクロプロセッサのためのＣ++やＪａｖａプログラミング言語で書かれている。以下のものが、バイトコードＩＬＯＡＤのためのテンプレート関数である。

```
void TemplateTable::iload(int n){
        assembler.movl(eax, address(n));
}
```

ＩＬＯＡＤメソッドは、Ｊａｖａ仮想マシンの場合には（for）ＴＥＭＰＬＡＴＥ＿ＴＡＢＬＥと呼ばれるクラスに対して定義される。ＩＬＯＡＤバイトコードは、オペランドスタックに整数をプッシュするので、同じ名前によるテンプレート関数は、整数であるパラメータを有する。ＭＯＶＬメソッドは、Ｎの値をレジスタに置くことによってオペランドスタックにＮの値をプッシュするＡＳＳＥＭＢＬＥＲオブジェクトのためのＣ++関数である。ＭＯＶＬは、Ｊａｖａ仮想マシンでは整数であるがｘ８６マイクロプロセッサでは倍長整数である３２ビット値を移すｘ８６アセンブリ言語命令に対応することを思い出そう。

【００５５】他の例として、以下のものは、バイトコードＩＡＤＤのためのテンプレート関数である。

```
void TemplateTable::iadd(){
        assembler.popl(edx);
        assembler.addl(eax, edx);
}
```

ＩＡＤＤメソッドは、Ｊａｖａ仮想マシンの場合には（for）ＴＥＭＰＬＡＴＥ＿ＴＡＢＬＥと呼ばれるクラスに対して定義される。ＩＡＤＤバイトコードのための期待される状態はＩＴＯＳであり、そのため、レジスタ（この例ではＥＡＸ）に格納されているオペランドスタックの最上位に整数があるべきである。まず、ＳＰ’ポインタ（これはｘ８６マイクロプロセッサでのＥＳＰポインタであることができる）によって指し示される値は、ＰＯＰＬメソッドを利用してポップされスタックから離される。今、われわれが説明しているスタックは、目的の（target）マイクロプロセッサ上のメモリに格納されている固有のスタック（図４（ｂ）および図４（ｃ）の左側を見よ）であることを理解するのは重要である。このため、ＳＰ’ポインタによって指し示されるデータ値は、レジスタＥＤＸに移され、次いで、ＳＰ’がデクレメントされる。

【００５６】この時点で、オペランドスタックの最上位は、レジスタＥＡＸに格納されて、オペランドスタックの次に高位のデータ値は、ＥＤＸに格納される。ＡＤＤＬメソッドは、ＥＡＸおよびＥＤＸに格納されている値を加え、その総和をＥＡＸに格納するアセンブリ言語命令に対応する。すると、ＥＡＸレジスタは、オペランドスタックの最上位として（for）適切なレジスタに所望の総和を格納して、これは、選択された関数ＩＡＤＤの実行の後に、図５のテンプレート表に明示されているように、インタプリタがＩＴＯＳ状態にあることを意味している。

【００５７】上で説明されたように、好ましい実施例においては、オブジェクトは、インタプリタにおいて利用

される各アセンブリ言語命令のためのメソッドを含むアセンブラのために具体例を上げて示される。単純にするため、メソッドの名前は、アセンブリ言語命令と同じである。アセンブラオブジェクトを使うことは、インタプリタの生成のためには有益であることが見いだされ、なぜなら、特別なアセンブラが利用されることを必要としないからである。ある実施例においては、テンプレート関数は、所望なコンピュータアーキテクチャのためのアセンブリ言語で書かれることができる。

【００５８】ステップ７１１では、システムは、エピローグのコンピュータコードを生成する。エピローグは、次の仮想マシン命令を実行するためにインタプリタを設定するコンピュータコードである。このため、エピローグは、前に説明されたインタプリタの進行ステップおよびディスパッチステップを実行する。

【００５９】最初に、プロローグのコンピュータコードは、次の仮想マシン命令をフェッチする。選択された仮想マシン命令の実行の後の現在のインタプリタの状態が（例えば、図５のテンプレート表のフィールド５０９から）わかるので、次の仮想マシン命令を実行するようにインタプリタ内の位置（ロケーション）を決定するために、次の仮想マシン命令を図６のディスパッチ表へのインデックス（またはオフセット）として利用できる。エピローグ内のコンピュータコードは、図８を参照して、より詳細に説明されるが、しかし一般的には、エピローグは、選択された仮想マシンおよび現在のインタープリタの状態に依存する。

【００６０】システムは、ステップ７０１においてインタプリタのためのコンピュータコードを生成するために、仮想マシン命令／インタプリタ状態がもっとあるかどうかを決定する。もしあれば、プロセスは、ステップ７０１に戻り、そして別の繰り返し（iteration）が実行される。

【００６１】ステップ７０５に戻って、選択された仮想マシン命令およびインタプリタ状態が非合法であると決定されたならば、システムは、ステップ７１５においてエラーを処理するためのコンピュータコードを生成できる。一般的に言って、エラーは非合法のバイトコードシーケンスである。好ましい実施例では、このエラーが生じたことをユーザに知らせる命令へジャンプするコンピュータコードが、インタプリタのために生成される。この方法で検出されたエラーの数は、バイトコード検証器（ヴェリファイア、verifier）によって検出されたエラーほど多くはないけれど、バイトコード検証器を使うことが要求されず、または、不可能であるならば、この方法は特に望ましい場合がある。ある実施例では、エラーをチェックすること及びステップ７０７およびステップ７１５を処理することは、省略されることができる。

【００６２】図８は、インタプリタのためのエピローグのコンピュータコードを生成するプロセスを示してい

る。エピローグのコンピュータコードは、図７のステップ７１１において生成されるが、しかしエピローグのコンピュータコードを生成する特定の実施例は、図８を参照して説明される。ステップ８０１においては、次の仮想マシン命令をフェッチするコンピュータコードが生成される。次の仮想マシン命令は、現在のバイトコードへ次のバイトコードポインタをインクリメントすることによって、またそれからそのバイトコードポインタによって指し示され、且つ参照されたバイトコードをフェッチすることによって、フェッチされることができる。仮想マシン命令のサイズは、Ｊａｖａバイトコードにおけるように変化するので、好ましい実施例においては、各バイトコードのサイズを格納するために表が利用されて、バイトコードポインタは、バイトコードを指し示すために、表内のサイズ分だけ（by）インクリメントされることができる。

【００６３】一旦、次の仮想マシン命令がフェッチされると、システムは、ステップ８０３において、ディスパッチ表へのオフセットを計算するためのコンピュータコードを生成する。そのオフセットは、図６のディスパッチ表の開始アドレスへ加えられるとき、次のバイトコードおよび現在のインタプリタ状態によってインデックス付けされたフィールドに帰着する数である。好ましい実施例においては、ディスパッチ表は、各インタプリタ状態に対して一つである、単一次元の５つの表（または補助表）である。インタプリタの現在の状態は、どの補助表（subtable）を使うかを決定する。補助表における各フィールドのサイズは、固定されたサイズ（例えば、４バイト）であることができ、そのため、オフセットを計算することは、次のバイトコード値を固定サイズ倍することを含む。ディスパッチ表が単一の２次元表である実施例においては、２次元配列にオフセットを計算する技術分野における当業者によく知られている多くの技法が、利用されることができる。さらに、この発明は表に限られるのではなく、リンクされたリスト、およびハッシュ表、等を含むような任意の数のデータ構造を利用して実現されることができる。

【００６４】ステップ８０５においては、システムは、ディスパッチ表内のオフセットのところにあるフィールドによって明示されている、インタプリタにおける位置（ロケーション）またはアドレスへジャンプするためのコンピュータコードを生成する。ディスパッチ表は、インタプリタ自身のコンピュータコード内にアドレスを格納するジャンプ表である。インタープリットしている中に、エピローグのコンピュータコードは、そのインタプリタの進行ステップおよびディスパッチステップを実行する。しかしながら、他の実施例は、プロローグに進行ステップを置き、エピローグにディスパッチステップを置くことができる。したがって、この発明は、ここで説明された特定の実現されたものに限定されない。

【００６５】上記のものは、本発明の好ましい実施例を記述した。概念的には、インタプリタ状態、ＩＴＯＳ、ＬＴＯＳ、ＦＴＯＳ、ＤＴＯＳ、ＶＴＯＳの各々に対して１つある、生成された５つの別個のインタプリタがあると考えることができる。しかしながら、実際には、仮想マシン命令／インタプリタ状態の組み合わせの多くは、非合法であり、それで、５つの別のインタプリタは生成されない。さらに、仮想マシン命令を実行するコンピュータコードは共有されることができ、それで、本発明に従うインタプリタは、伝統的なインタプリタよりもサイズがずっと大きくなることはない。仮想マシン命令を実行するコンピュータコードがどのように共有されることができるかをよりはっきりと見るために、見本のバイトコードのためのコンピュータコードがインタプリタのためにどのように生成されることができるかを説明することは、読者に役に立つかもしれない。

【００６６】例

図７を参照して説明されたように、可能な仮想マシン命令およびインタプリタ状態の組み合わせ（combination）の間を次々と（through）循環し、またはループすることによって、インタプリタを生成することができる。インタプリタが生成されるとき、異なった仮想マシン命令およびインタプリタ状態の結合（combination）のためのエントリポイントまたはジャンプポイントを保持するためにインタプリタの実行中にディスパッチ表が利用された状態で、各仮想マシン命令のためのコンピュータコードのセクションが生成される。したがって、生成されたインタプリタは、ディスパッチ表、および異なる仮想マシン命令を実行する（またはエラーを処理する）一連のコンピュータコードのセクションを含むことができる。

【００６７】この例の場合、ＩＡＤＤバイトコードを実行するコンピュータコードが、どのように生成されるかが記述される。図９は、ＩＡＤＤバイトコードに関連する（pertain）ディスパッチ表９０１の部分を示している。ディスパッチ表の構造は、図６を参照して説明されたものと同じである。要するに、ディスパッチ表は、仮想マシン命令９０３によってインデックス付けされ、またインタプリタの各状態に対して１フィールドである、フィールド９０５、９０７、９０９、９１１、および９１３を含む。

【００６８】ＩＡＤＤバイトコードのための１０進数の値は、括弧内に示されているように９６である。インタプリタがＩＴＯＳ状態ならば、ポインタA_Iは、ＩＡＤＤバイトコードを実行するインタプリタにおける位置（ロケーション）またはアドレスを差し指す。同様に、インタプリタが、それぞれ、ＬＴＯＳ、ＦＴＯＳ、ＤＴＯＳ、ＶＴＯＳの状態にあるとき、A_L、A_F、A_D、A_Vは、ＩＡＤＤバイトコードを実行するインタプリタにおける位置（ロケーション）またはアドレスを差し指す。

【００６９】ディスパッチ表９０１におけるポインタは、インタプリタのために生成されたコンピュータコードの一連のセクション内のアドレスを差し指す。図１０は、インタプリタのために生成されたコンピュータコードの一連のセクションを示す。コンピュータコードの各セクションは、特定のバイトコードを実行する。セクション１００３は、ＩＡＤＤバイトコードの実行のためのコンピュータコードを含む。示されているように、セクション１００３は、２つのＰＯＰＬ命令と１つのＡＤＤ

```
ITOS
POPL EDX
ADDL EAX,EDX
<DISPATCH/ADVANCE>
```

【００７２】簡単のために、ディスパッチステップおよび進行ステップを実行するコンピュータコードは、明示的に示されていない。示されるように、コンピュータコードの２つのセクション間の唯一の違いは、インタプリタがＶＴＯＳ状態にあるとき、付加的な命令があることである。ＰＯＰＬ ＥＡＸは値をポップし固有のマシンのスタックから離して、それをレジスタＥＡＸに置く。この命令は、（図７のステップ７０７を見よ）ＩＡＤＤバイトコードのために期待された状態であるＩＴＯＳへインタプリタがＶＴＯＳ状態から遷移する（シフトする、shift）ために、プロローグとして生成された。

【００７３】ＰＯＰＬ ＥＤＸ命令およびＡＤＤＬ命令は、テンプレート表（前の例と図７のステップ７０９を見よ）においてアクセスされたテンプレート関数ＩＡＤＤ（）によって生成された。さらに、ディスパッチステップや進行ステップを実現するコンピュータコードは、エピローグのコンピュータコード（図７のステップ７１１を見よ）である。コードの各セクションは、最初のアセンブリ言語命令のみによって異なるので、コンピュータコードの単一のセクションにアクセスするためにポインタが利用されることができる。

【００７４】故に、インタプリタがＩＴＯＳ状態またはＶＴＯＳ状態のいずれかにあるとき、セクション１００３は、ＩＡＤＤバイトコードを実行するためのコンピュータコードを含んでいる。図１０に示されるように、図９のディスパッチ表９０１からのポインタA_Vは、セクション１００３の最初の命令を差し指し、このため、ＶＴＯＳ状態からＩＴＯＳ状態へインタプリタを置く（put）初期命令が実行される。インタプリタがＩＴＯＳ状態にあるので、ディスパッチ表からのポインタA_Iは、セクション１００３における２番目の命令を差し指す。示されているように、インタプリタがＶＴＯＳ状態またはＩＴＯＳ状態にあろうとなかろうと、ポインタA_Iによって明示されるコンピュータコードは、ＩＡＤＤバイトコードを実行するようにインタプリタに指示する。

【００７５】ディスパッチ表９０１において、ＩＡＤＤ

Ｌ命令を含む。これらの命令は、ｘ８６マイクロプロセッサのためのアセンブリ言語（または、固有のマシン命令）であり、そして以下のように生成された。

【００７０】インタプリタ生成中に、インタプリタがＩＴＯＳまたはＶＴＯＳ状態にあるとき、ＩＡＤＤバイトを実行するために、アセンブリ言語の命令の以下のセクションが生成されることができる。

【００７１】

```
VTOS
POPL EAX
POPL EDX
ADDL EAX,EDX
<DISPATCH/ADVANCE>。
```

バイトコードに対するＬＴＯＳ、ＦＴＯＳ、ＤＴＯＳ状態は、そのバイトコードに対する非合法の状態を表わしている。故に、ポインタA_L、A_F、A_Dポインタは、エラーを処理する図１０におけるコンピュータコードのセクション１００５を差し指す。セクション１００５におけるコンピュータコードは、典型的には、インタプリタが非合法状態（図７のステップ７１５も見よ）に置かれたことをユーザに示している。簡単のために、エラーを処理するコンピュータコードの１つのセクションが示されているが、しかしながら、コンピュータコードの２以上のセクション（または、エラーチェックが望まれていないならば、なし）が利用されることができる。加えて、仮想マシン命令の実行またはエラーを処理するコンピュータコードのセクションが、任意の順序に配置されることができる。

【００７６】例を説明してきたが、図１１に示されるように１つの実施例に従って、インタプリタを用いて仮想マシン命令の実行するプロセスを説明することは役に立つことがある。示されたプロセスは、ここで説明されるように生成されたインタプリタにおいて仮想マシン命令を実行するために利用されることができる。しかしながら、そのプロセスは、他の方式において生成されたインタプリタを用いて利用されることができ、そのため、記載されたインタプリタ生成が、インタプリタを実現することを限定するものとして受け取られるべきではない。

【００７７】ステップ１１０１において、コンピュータシステムは、インタプリタを期待された状態に置き、ここで、期待された状態とは、選択された仮想マシン命令が実行される前の、期待されるインタプリタの状態である。他の実施例においては、期待された状態は、選択された仮想マシン命令を実行するインタプリタにおけるコンピュータコード（例えば、テンプレート関数によって生成されたコンピュータコード）が行われる前の、期待されるそのインタプリタ状態である。ステップ１１０１は、インタプリタの実行中に生じて、またインタプリタの生成中に図７のステップ７０７において生成されたプ

ロローグのコンピュータコードに対応する。システムが期待された状態にあるならば、このステップは、省略されることができる。

【００７８】ステップ１１０３において、システムは、選択された仮想マシン命令を実行する。このステップはインタプリタ実行中に生じ、インタプリタの生成中に図７のステップ７０９においてテンプレート関数によって生成されたコンピュータコードに対応する。

【００７９】一旦、選択された仮想マシン命令が実行されたら、システムは、ステップ１１０５において次の仮想マシン命令をフェッチする。次の仮想マシン命令を利用して、システムは、ステップ１１０７においてディスパッチ表へのオフセットを計算する。選択された仮想マシン命令が実行された後でのインタプリタの現在の状態は知られている。したがって、次の仮想マシン命令と共に現在の状態が利用され、次の仮想マシン命令を実行するためにインタプリタにおける位置を明示するディスパッチ表へのオフセットを計算することができる。各インタプリタ状態に対して１つある、多重の単一次元の補助表として、ディスパッチ表が実現される好ましい実施例においては、現在の状態は、補助表を明示して、またオフセットは、次の仮想マシン命令を利用して（例えば、仮想マシン命令＊固定サイズ）計算される。

【００８０】ステップ１１０９において、システムはディスパッチ表におけるオフセットのところのフィールドに格納されたインタプリタにおけるアドレスまたは位置（ロケーション）にジャンプする。フィールドは、次の仮想マシン命令を実行するインタプリタの位置（ロケーション）へのポインタを含むことができる。このため、そのジャンプをすると、次の仮想マシン命令に対してステップ１１０１へシステムが戻る。

【００８１】ステップ１１０５、１１０７、１１０９は、インタプリタの実行中に生じて、またインタプリタの生成中に図７のステップ７１１においてエピローグのコンピュータコードによって生成されたコンピュータコードに対応する。図１１で示された発明の実施例の場合、明示的なエラーチェックは要求されていないことを注目すべきである。その代わりに、エラーがあったら、システムは、ステップ１１０９においてエラーを処理するためのコンピュータコードにジャンプする。故に、エラーをチェックすることは、効率に関して大きな影響なしに達成されることができる。

【００８２】結論

上記は、本発明の好ましい実施例の完全な記述である一方で、様々な代替物、変形物、および均等物が使用されることができる。本発明は、上で記述された実施例に適切な変形を成すことによって、同等に利用可能なものになることは明らかである。例えば、記述された実施例は、Ｊａｖａ仮想マシンをインタープリットするバイトコードの効率を上昇することに関連しているが、しかし、現在の発明の方針は、容易に他のシステムおよび言語に適用されることができる。従って、上記の記載は、等価物の全範囲とともに、書かれた特許請求の範囲（impendedclaims）の集まりおよび境界によって定義される発明の範囲に限定してものとして受け取るべきではない。

【図面の簡単な説明】

【図１】図１は、本発明の実施例のソフトウェアを実行するために利用されることができるコンピュータシステムの例を示す概略図である。

【図２】図２は、図１のコンピュータシステムのシステムブロック図である。

【図３】図３は、Ｊａｖａソースコードプログラムがどのように実行されるかを示す概念図である。

【図４】図４（ａ）は、スタックを示す概念図であり、図４（ｂ）は、本発明のオペランドスタックを示す概念図であって、ここでオペランドスタックの最上位に関する値はレジスタに格納されている。図４（ｃ）は、本発明のオペランドスタックを示す概略図であり、ここでマルチプルレジスタ、および異なったデータタイプを格納するレジスタはオペランドスタックの最上位に関する値を格納することができる。

【図５】図５は、インタプリタ状態、およびテンプレート関数を組織化するために、インタプリタ発生中に使用されたテンプレート表図である。

【図６】図６は、インタプリタ実行中に生成されるディスパッチ表図であって、この図面は、インタプリタの実行フローを指示するために利用された、インタプリタ内のロケーションへのポインタを格納する。

【図７】図７は、１つ以上のレジスタに格納されるオペランドスタックの最上位に関する値のデータタイプを示すためにインタプリタの状態を利用しているインタプリタを生成するプロセス図である。

【図８】図８は、インタプリタの進行ステップおよびディスパッチステップを実行するエピローグのコンピュータコードを生成するプロセス図である。

【図９】図９は、バイトコードＩＡＤＤを実行するための図６のディスパッチ表図の部分を示す概念図である。

【図１０】図１０は、仮想マシン命令を実行し、またエラーを処理するインタプリタのコンピュータコードのセクションを示す概念図である。

【図１１】図１１は、本発明の実施例に従う、インタプリタを用いて仮想マシン命令を実行するプロセスを示すプロセス図である。

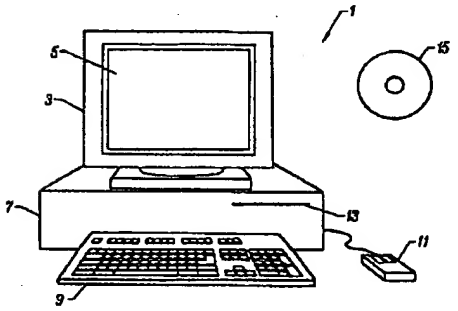
【符号の説明】

１…コンピュータシステム、３…ディスプレイ、５…スクリーン、７…キャビネット、９…キーボード、１１…マウス、１３、１５…ＣＤ－ＲＯＭドライブ、５１…中央プロセッサ、５３…システムメモリ、５５…固定記憶装置、５７…リムーバブル記憶装置、５９…ディスプレ

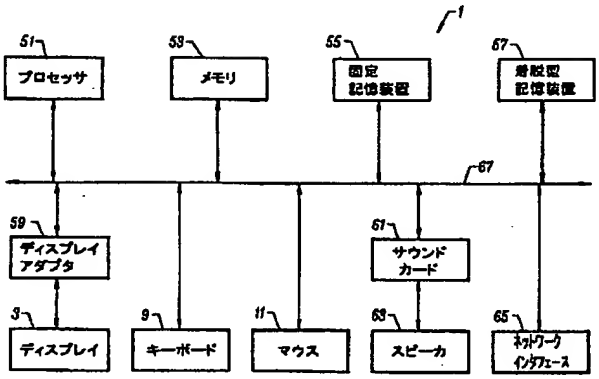
イアダプタ、６１…サウンドカード、６３…スピーカ、６５…ネットワークインタフェース、１０１…Ｊａｖａソースコード、１０３…バイトコードコンパイラ、１０５…Ｊａｖａクラスファイル、１０７…Ｊａｖａ仮想マシン、２０１、３０１、４０１…オペラントスタック、２０３、３０３、４０３…データ、２０５、３０７、４１１…スタックポインタ（ＳＰ）、３０５、４０５、４０７、４０９…レジスタ

５０１…テンプレート表（データ構造）、５０３…バイトコード、５０５、５０７、５０９…フィールド、６０１…ディスパッチ表、６０３…仮想マシン命令、６０５、６０７、６０９、６１１、６１３…フィールド、９０１…ディスパッチ表、９０３…仮想マシン命令、９０５、９０７、９０９、９１１、９１３…フィールド、１００３、１００５…セクション、

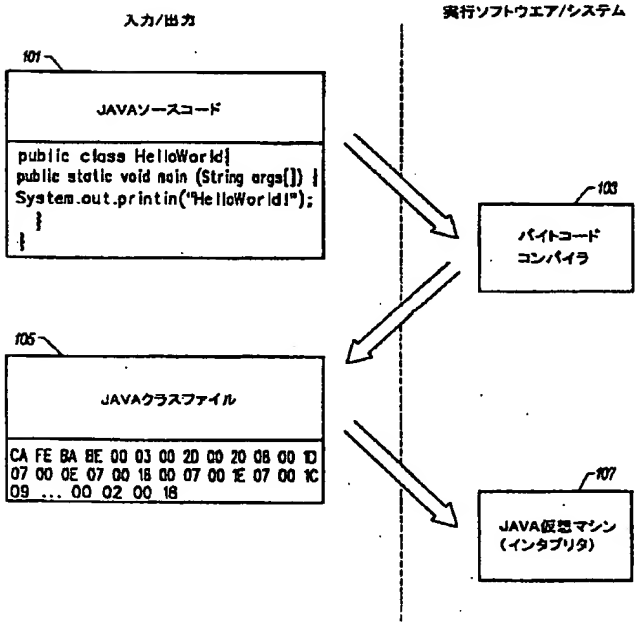
【図１】



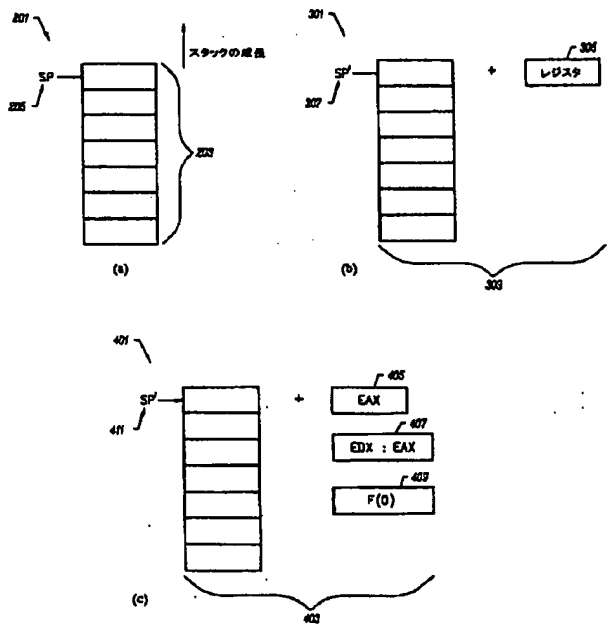
【図２】



【図３】



【図４】



【図５】

テンプレート表

バイトコード	実行前の状態	テンプレート関数	実行後の状態
0	<任意>	NOP( )	<同じ>
...	...		
21	VTOS	ILOAD( )	ITOS
22	VTOS	LLOAD( )	LTOS
23	VTOS	FLOAD( )	FTOS
24	VTOS	DLOAD( )	DTOS
...	...		
54	ITOS	ISTORE( )	VTOS
55	LTOS	LSTORE( )	VTOS
56	FTOS	FSTORE( )	VTOS
57	DTOS	DSTORE( )	VTOS
...			
96	ITOS	IADD ( )	ITOS
97	LTOS	LADD ( )	LTOS
98	FTOS	FADD ( )	FTOS
99	DTOS	DADD ( )	DTOS
...	...		

501, 503, 505, 507, 509

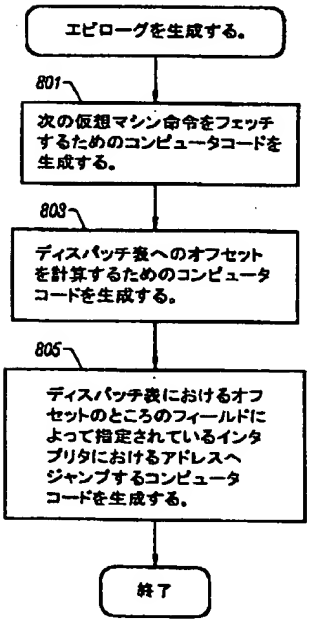
【図６】

ディスパッチ表

バイトコード	ITOS	LTOS	FTOS	DTOS	VTOS
0	<PTR>	<PTR>	<PTR>	<PTR>	<PTR>
1	<PTR>	<PTR>	<PTR>	<PTR>	<PTR>
2	<PTR>	<PTR>	<PTR>	<PTR>	<PTR>
3	<PTR>	<PTR>	<PTR>	<PTR>	<PTR>
4	<PTR>	<PTR>	<PTR>	<PTR>	<PTR>
...					

601, 603, 605, 607, 609, 611, 613

【図８】



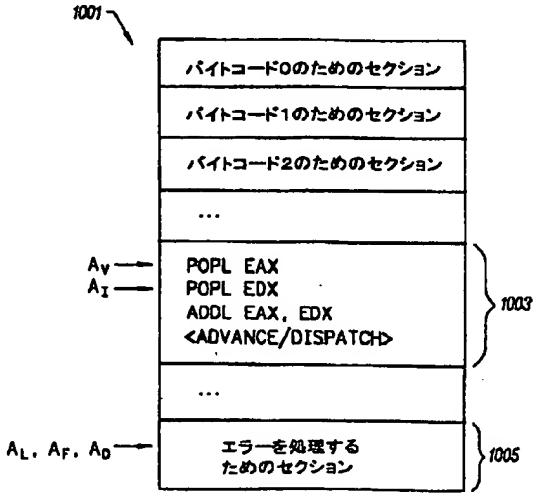
【図９】

ディスパッチ表

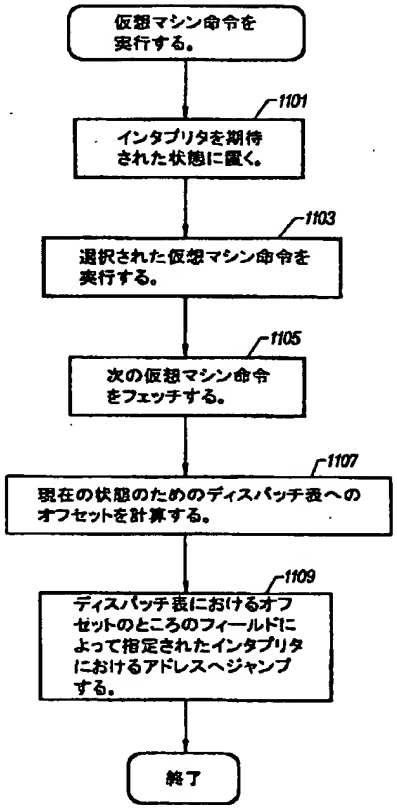
バイトコード	ITOS	LTOS	FTOS	DTOS	VTOS
...	...				
96 (IADD)	A_I	A_L	A_F	A_D	A_V
...	...				

901, 903, 905, 907, 909, 911, 913

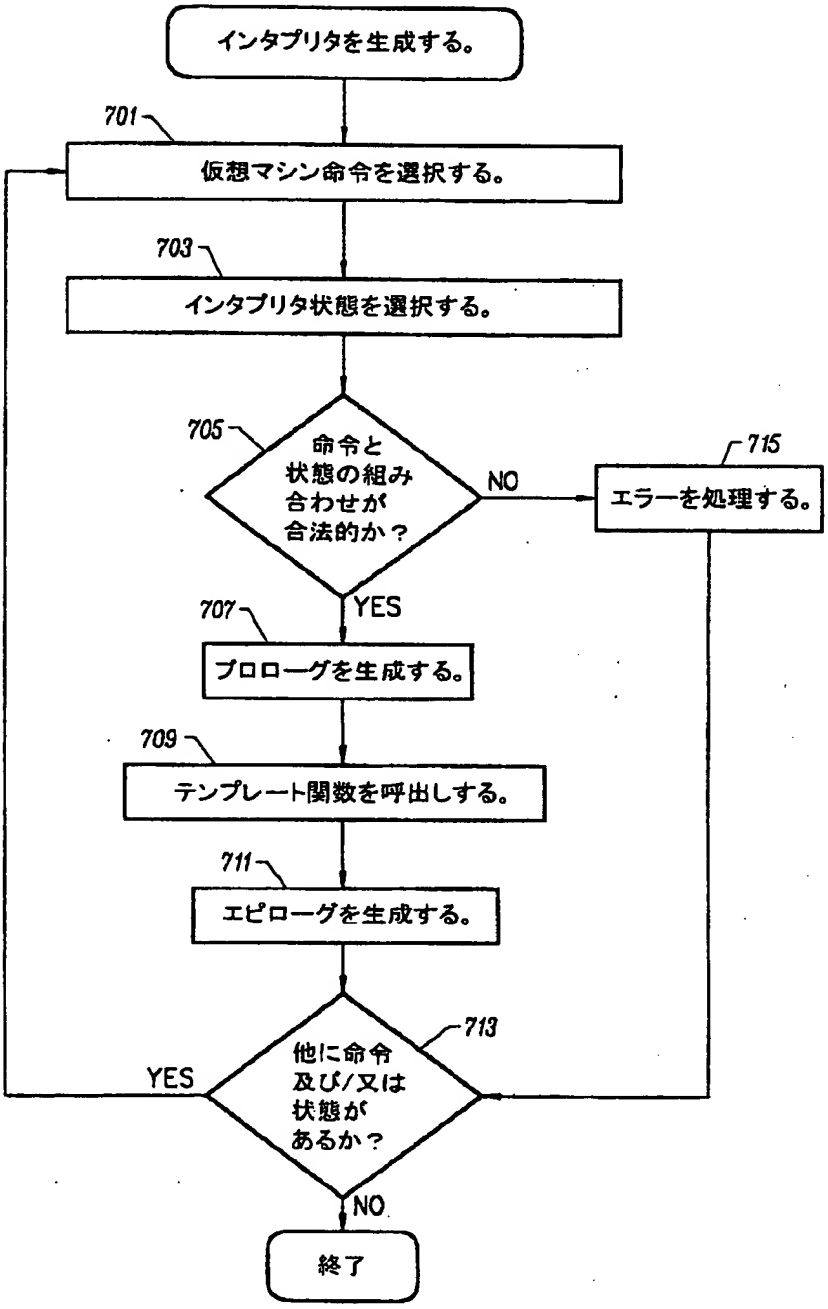
【図１０】



【図１１】



【図７】


インタプリタを生成する。
701
仮想マシン命令を選択する。
703
インタプリタ状態を選択する。
705
命令と状態の組み合わせが合法的か？
NO
715
エラーを処理する。
YES
707
プロローグを生成する。
709
テンプレート関数を呼出しする。
711
エピローグを生成する。
713
他に命令及び/又は状態があるか？
YES
NO
終了

【外国語明細書】

1. Title of Invention

INTERPRETER GENERATION AND IMPLEMENTATION UTILIZING INTERPRETER STATES AND REGISTER CACHING

2. Claims

1. In a computer system including a plurality of registers, a method for implementing an interpreter including an operand stack having a top, the method comprising:

storing a value for the top of the operand stack in at least one register of the plurality of registers; and

utilizing a state of the interpreter to indicate a data type of the value for the top of the operand stack that is stored in the at least one register.

2. The method of claim 1, wherein the data type is selected from the group consisting of integer, long integer, single-precision floating point, and double-precision floating point.

3. The method of one of claims 1 or 2, wherein the data type is void to indicate that the value for the top of the operand stack is not currently stored in the at least one register.

4. The method of one of claims 1-3, wherein an instruction that utilizes the top of the operand stack accesses the at least one register.

5. The method of one of claims 1-4, wherein the plurality of registers includes at least two registers that are used for storing values of different data types.

6. The method of one of claims 1-5, wherein the data type indicated by the state of the interpreter specifies the at least one register of the plurality of registers that stores the top of the operand stack.

7. The method of one of claims 1-6, wherein the interpreter is a Java virtual machine.

8. In a computer system including a plurality of registers, a method for generating an interpreter that stores a value for a top of an operand stack in the at least one register, the method comprising:

selecting a virtual machine instruction to be interpreted by the interpreter;

selecting a state of the interpreter, wherein the state of the interpreter indicates a data type of the value for the top of the operand stack that is stored in at least one register of the plurality of registers;

generating computer code for the interpreter to put the interpreter in an expected state for the selected virtual machine instruction if the selected state differs from the expected state; and

generating computer code for the interpreter to execute the selected virtual machine instruction.

9. The method of claim 8, wherein the expected state for the selected virtual function is obtained by accessing a table indexed by virtual machine instructions that stores expected states of the interpreter before execution of the virtual machine instructions.

10. The method of claim 9, wherein the table stores current states of the interpreter after execution of the virtual machine instructions.

11. The method of claim 9, wherein the table stores pointers to functions that generate computer code for the interpreter to execute the virtual machine instructions.

12. The method of one of claims 8-11, wherein the computer code for the interpreter to execute the selected function is generated by calling a function specified in a table indexed by virtual machine instructions that stores pointers to functions that generate computer code for the interpreter to execute the virtual machine functions.

13. The method of one of claims 8-12, further comprising generating computer code for the interpreter to fetch the next virtual machine instruction.

14. The method of one of claims 8-13, further comprising generating computer code for the interpreter to jump to a location in the interpreter that executes the next virtual machine instruction for a current state of the interpreter after execution of the selected virtual machine instruction.

15. The method of one of claims 8-14, wherein the location in the interpreter handles an error if the selected virtual machine instruction is illegal for the selected state.

16. The method of one of claims 8-15, wherein the interpreter is a Java virtual machine.

3. Detailed Description of Invention

The Java virtual machine is commonly implemented as an software interpreter. Conventional interpreters decode and execute the virtual machine instructions of an interpreted program one instruction at a time during execution. Compilers, on the other hand, decode source code into native machine instructions prior to execution so that decoding is not performed during execution. Because conventional interpreters decode each instruction before it is executed repeatedly each time the instruction is encountered, execution of interpreted programs is typically quite slower than compiled programs because the native machine instructions of compiled programs can be executed on the native machine or computer system without necessitating decoding.

As a software interpreter must be executing in order to decode and execute an interpreted program, the software interpreter consumes resources (e.g., memory) that will therefore no longer be available to the interpreted program. This is in stark contrast to compiled programs that execute as native machine instructions so they may be directly executed on the target computer and are therefore generally free to utilize more resources than interpreted programs.

Accordingly, there is a need for new techniques for increasing the execution speed of computer programs that are being interpreted. Additionally, there is a need to provide interpreters that are efficient in terms of the resources they require.

SUMMARY OF THE INVENTION

In general, some embodiments of the present invention provide innovative systems and methods for increasing the execution speed of computer programs executed by an interpreter. The interpreter includes an operand stack that is utilized to execute the virtual machine instructions. The value for the top of the operand stack is stored in one or more registers which allows the execution speed of stack-based virtual machine instructions to be increased. A state of the interpreter is utilized to indicate the data type of the value for the top of the operand stack stored in the one or more registers. With the invention, the programs may be interpreted in a more efficient manner utilizing registers. Additionally, the size of the interpreter may kept small which allows more resources to be available for the interpreted program. Several embodiments of the invention are described below.

In one embodiment, a computer implemented method for implementing an interpreter including an operand stack is provided. A value for the top of the operand stack is stored in at least one register of the computer instead of on the stack. Many conventional computers have registers for storing different data types. Accordingly, the value for the top of the stack is stored in one or more registers appropriate for its data type and the state of the interpreter is utilized to indicate the data type of the value for the top of the operand stack that is stored in the one or more registers. In preferred embodiments, the interpreter is a Java virtual machine and the states of the interpreter may include integer, long integer, single-precision floating point, and double-precision floating point.

In another embodiment, a computer implemented method for generating an interpreter that stores a value for the top of an operand stack in one or more registers is provided. The state of the interpreter indicates a data type of the value for the top of the operand stack that is stored in the one or more registers. In order to generate the interpreter, the computer may loop through all the possible virtual instructions and states of the interpreter. In each iteration, a virtual machine instruction and a state of the interpreter may be selected. If the selected state differs from the state of the interpreter that is expected prior to the execution of the selected virtual machine instructions, computer code for the

interpreter is generated to put the interpreter in the expected state. Once it is known that the interpreter is in the expected state prior to the execution of the selected virtual machine instruction, computer code for the interpreter is generated to execute the selected virtual machine instruction. The expected state for the selected virtual machine instruction may be obtained by accessing a table indexed by virtual machine instructions that stores expected states of the interpreter before execution of the virtual machine instructions and current states of the interpreter after execution of the virtual machine instructions. Additionally, the computer code for the interpreter to execute the selected virtual machine instruction may be generated by calling a function specified in the table.

In another embodiment, a data structure stored by a computer readable medium for an interpreter of virtual machine instructions is provided. The data structure is a table indexed by virtual machine instructions and having multiple fields. In one field of the table, expected states of the interpreter before execution of the virtual machine instructions are stored. In another field of the table is stored current states of the interpreter after execution of the virtual machine instructions. Additionally, a field of the table may be utilized to store pointers to functions that generate computer code for the interpreter to execute the virtual machine instructions. In a preferred embodiment, a state of the interpreter indicates the data type of a value for the top of an operand stack of the interpreter that is stored in one or more registers.

In another embodiment, a data structure stored by a computer readable medium for an interpreter of virtual machine instructions is provided. The data structure is a table indexed by virtual machine instructions and having multiple fields, each field being associated with a state of the interpreter and storing a pointer to a location in the interpreter that executes the indexed virtual machine instructions. The state of the interpreter may indicate the data type of a value for the top of an operand stack of the interpreter that is stored in one or more registers. In preferred embodiments, the state of the interpreter may be integer, long integer, single-precision floating point, and double-precision floating point.

Other features and advantages of the invention will become readily apparent upon review of the following detailed description in association with the accompanying drawings.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Definitions

Machine instruction - An instruction that directs a computer to perform an operation specified by an operation code (opcode) and optionally one or more operand.

Virtual machine instruction - A machine instruction for a software emulated microprocessor or computer architecture (also called virtual code).

Native machine instruction - A machine instruction that is designed for a specific microprocessor or computer architecture (also called native code).

Class - An object-oriented data type that defines the data and methods that each object of a class will include.

Function - A software routine (also called a subroutine, procedure, member function, and method).

Operand stack - A stack utilized to store operands for use by machine instructions during execution.

Bytecode pointer (BCP) - A pointer that points to the current Java virtual machine instruction (e.g., bytecode) that is being executed.

Program counter (PC) - A pointer that points to the machine instruction (typically native) of the interpreter that is being executed.

Interpreter - A program in software or hardware that typically translates and then executes each instruction in a computer program.

Interpreter generator - A program in software or hardware that generates an interpreter.

Overview

In the description that follows, the present invention will be described in reference to a preferred embodiment that implements a Java virtual machine for executing Java virtual machine instructions (bytecodes). In particular, examples will be described including native

machine instructions of IBM personal computers (Intel x86 microprocessor architectures). However, the invention is not limited to any particular language, computer architecture, or specific implementation. Therefore, the description of the embodiments that follow is for purposes of illustration and not limitation.

Fig. 1 illustrates an example of a computer system that may be used to execute the software of an embodiment of the invention. Fig. 1 shows a computer system 1 which includes a display 3, screen 5, cabinet 7, keyboard 9, and mouse 11. Mouse 11 may have one or more buttons for interacting with a graphical user interface. Cabinet 7 houses a CD-ROM drive 13, system memory and a hard drive (see Fig. 2) which may be utilized to store and retrieve software programs incorporating computer code that implements the invention, data for use with the invention, and the like. Although the CD-ROM 15 is shown as an exemplary computer readable storage medium, other computer readable storage media including floppy disk, tape, flash memory, system memory, and hard drive may be utilized. Additionally, a data signal embodied in a carrier wave (e.g., in a network including the Internet) may be the computer readable storage medium.

Fig. 2 shows a system block diagram of computer system 1 used to execute the software of an embodiment of the invention. As in Fig. 1, computer system 1 includes monitor 3 and keyboard 9, and mouse 11. Computer system 1 further includes subsystems such as a central processor 51, system memory 53, fixed storage 55 (e.g., hard drive), removable storage 57 (e.g., CD-ROM drive), display adapter 59, sound card 61, speakers 63, and network interface 65. Other computer systems suitable for use with the invention may include additional or fewer subsystems. For example, another computer system could include more than one processor 51 (i.e., a multi-processor system), or a cache memory.

The system bus architecture of computer system 1 is represented by arrows 67. However, these arrows are illustrative of any interconnection scheme serving to link the subsystems. For example, a local bus could be utilized to connect the central processor to the system memory and display adapter. Computer system 1 shown in Fig. 2 is but an example of a computer system suitable for use with the invention. Other computer architectures having different configurations of subsystems may also be utilized.

Typically, computer programs written in the Java programming language are compiled into bytecodes or Java virtual machine instructions which are then executed by a Java virtual machine. The bytecodes are stored in class files which are input into the Java virtual machine for interpretation. Fig. 3 shows a progression of a simple piece of Java source code through execution by an interpreter , the Java virtual machine.

Java source code 101 includes the classic Hello World program written in Java. The source code is then input into a bytecode compiler 103 which compiles the source code into bytecodes. The bytecodes are virtual machine instructions as they will be executed by a software emulated computer. Typically, virtual machine instructions are generic (i.e., not designed for any specific microprocessor or computer architecture) but this is not required. The bytecode compiler outputs a Java class file 105 which includes the bytecodes for the Java program.

The Java class file is input into a Java virtual machine 107. The Java virtual machine is an interpreter that decodes and executes the bytecodes in the Java class file. The Java virtual machine is an interpreter, but is commonly referred to as a virtual machine as it emulates a microprocessor or computer architecture in software (e.g., the microprocessor or computer architecture that may not exist in hardware).

An interpreter may execute a bytecode program by repeatedly executing the following steps:

Execute - execute operation of the current bytecode

Advance - advance bytecode pointer to next bytecode

Dispatch - fetch the bytecode at the bytecode pointer and jump to the implementation (i.e., execute step) of that bytecode.

The execute step implements the operation of a particular bytecode. The advance step increments the bytecode pointer so that it points to the next bytecode. Lastly, the dispatch step fetches the bytecode at the current bytecode pointer and jumps to the piece of native machine code that implements that bytecode. The execution of the execute-advance-dispatch sequence for a bytecode is commonly called an "interpretation cycle."

Although in a preferred embodiment, the interpreter utilizes the interpretation cycle described above, many other interpretation cycles may be utilized in conjunction with the present invention. For example, an interpreter may perform dispatch-execute-advance interpretation cycles or there may be more or fewer steps in each cycle. Accordingly, the invention is not limited to the embodiments described herein.

Java Virtual Machine Implementation and Generation

The virtual machine instructions for the Java virtual machine include stack-based instructions. Thus, one or more of the operands of the virtual machine instructions may be stored in an operand stack. Before describing the operand stack, it may be beneficial to discuss a general stack.

Fig. 4A shows a stack 201 that stores data 203. With the stack, a data value may be "pushed" onto the top of the stack. Alternatively, a data value may be "popped" off the top of the stack. Conceptually, only the top of the stack may be accessed so the stack is known as a first in, first out (FIFO) data structure meaning that the data that was most recently pushed onto the stack will be the first data to be popped off the stack. A stack pointer (SP) 205 points to the top of stack 201. Thus, the SP will change as data values are pushed onto and popped off the stack. For simplicity, the stacks described herein are shown and described as growing upwards in memory; however, this is just a graphical representation and those of skill in the art will readily recognize that a stack may be shown and/or implemented in many other ways (e.g., growing downwards in memory).

The virtual machine instructions for the Java virtual machine call for an implementation of an operand stack that may be similar to the stack shown in Fig. 4A. More specifically, the operand stack in the Java virtual machine is utilized to store operands for use by the bytecodes during execution. The following is an example that may help illustrate how an operand stack is utilized in the Java virtual machine.

Assume that the Java source code includes a statement X:=A+B, where X, A and B are integer variables. This statement may be compiled into the following bytecodes:

1. ILOAD A

2. ILOAD B

3. IADD

4. ISTORE X

It should be noted that the "I" preceding each bytecode indicates that the data type of the values manipulated by the bytecodes are integers. There are corresponding bytecodes for other data types which are designated by an "L" for long integer, "F" for single-precision floating point, and "D" for double-precision floating point. In the Java virtual machine, the integer data type is 32 bits, long integer data type is 64 bits, the single-precision floating point data type is 32 bits, and the double-precision floating point data type is 64 bits. As will be illustrated below, the size of these data types in the virtual machine may not be the same as in the computer system implementing the virtual machine (e.g., the standard integer data type on an IBM personal computer is 16 bits and there is no direct support for 64 bit long integers), so it is important to make a distinction between the Java virtual machine instructions and the native machine instructions that direct the computer on which the virtual machine is implemented.

The first ILOAD bytecode loads the value of variable A onto the operand stack. Similarly, the second ILOAD bytecode loads the value of variable B onto the operand stack. The bytecode IADD pops two data values off the operand stack, adds the two values and pushes the sum onto the operand stack. As may be expected, the ISTORE bytecode pops a data value off the operand stack, the sum in this case, and stores it in variable X. This simple example illustrates conceptually how Java bytecodes utilize an operand stack.

Some embodiments of the present invention take advantage of the fact that oftentimes when a data value is pushed onto the operand stack, it will subsequently be popped off the operand stack (with or without intervening bytecodes that do not modify the top of the operand stack). Typically, the operand stack is implemented in memory but with some embodiments of the present invention, the value for the top of the operand stack is stored in one or more registers. As registers have a faster access time than memory, access time for the top of the operand stack may be decreased resulting in faster interpretation of the interpreted program.

Fig. 4B shows an operand stack 301 that stores data 303. Data 303 includes both data values in memory and a data value for the top of the operand stack stored in a register 305. A stack pointer' (SP') 307 points to the data value conceptually just below the top of operand stack 301. It should be noted that this discussion focuses on a single operand stack. However, there may be multiple operand stacks in a single computer system (e.g., for different threads and methods) which may implemented according to the present invention.

Conventional computer systems typically have many different registers, with certain registers being better suited to store specific data types. For example, a computer may have registers that are 32 bits wide and registers that are 64 bits wide. If an integer on this hypothetical computer is a 32 bit quantity while a long integer is a 64 bit quantity, it would be more efficient to store the data values in a register that matches the size of the data value. Furthermore, computer systems may have registers that are designed to store specific data types like single-precision floating point or double-precision floating point.

More specifically, the IBM personal computers (Intel x86 architectures) include many different types of registers. There are 32 bit registers (e.g., EAX) and floating point registers (e.g., F(0)). Additionally, some of the data types of Intel x86 microprocessors have different sizes then their counterparts in the Java virtual machine. For example, a standard integer data type is 16 bits wide and a long integer data type is 32 bits wide in an Intel 80386 microprocessor. These data types are half the size of their Java virtual machine counterparts. Accordingly, in preferred embodiments where the Java virtual machine is implemented on an x86 machine, an integer in the Java virtual machine is the same size as a long integer in the x86 machine (i.e., both are 32 bit quantities).

As mentioned previously, it is important to keep in mind whether an instruction is a virtual machine instruction or a native machine instruction. Although the Java virtual machine instructions look similar to assembly language, fortunately there is an easily recognizable difference. In the bytecodes described herein for the Java virtual machine, the data type for which the bytecodes pertain precedes the instruction (e.g., ISTORE where the "I" designates integer). This is in stark contrast to the assembly code (or native machine

instructions) for the x86 microprocessors where the data type follows the instructions (e.g., POPL where the "L" designates long integer). Although this will likely not be true for all embodiments of the invention, it is hoped that this distinction will aid the reader's understanding of the preferred embodiments described herein.

Returning briefly to Fig. 4B, there is a problem since there is only one register shown and operand stack 301 may be utilized to store different data types so it would be desirable to have different registers available to store the value for the top of the operand stack. Fig. 4C shows an operand stack 401 for a Java virtual machine implemented on an x86 microprocessor . Operand stack 401 stores data 403 which includes both data values in memory and a data value for the top of the operand stack stored in one of registers 405, 407 or 409. In a preferred embodiment, register 405 is a 32 bit register (EAX) for storing virtual machine integers for the top of the operand stack. Register 407 is a combination of two 32 bit registers (EDX:EAX) for storing virtual machine long integers for the top of the operand stack. Register 409 is a 64 bit floating point register (F(0)) for storing both single-precision floating point and double-precision floating point. The designation of specific registers is provided to better illustrate the invention; however, it should be understood that the present invention is not limited to any specific registers or computer architectures.

A stack pointer' (SP') 411 points to the data value conceptually just below the top of operand stack 401. Now that there is more than one register that may be storing the value for the top of the operand stack, it would be desirable to know which register or registers stored this value. One technique would be to store a value in memory or a register indicating the data type of the value on the top of the operand stack. Accordingly, this could be accessed to determine the right register or registers storing the top of the operand stack.

Although this technique may work, it has some significant drawbacks that may make it unsatisfactory. For example, the extra determination of the data type of the top of the operand stack may offset the performance increase of utilizing registers to store the top of the operand stack.

With some embodiments of the present invention, the interpreter operates in multiple states. Each state indicates the data type of the value of the top of the operand stack stored

in the one or more registers. The state is an inherent quality of the interpreter at any point in time so a determination of the data type of the top of the operand stack is not required.

In a preferred embodiment, the interpreter may be in one of five different states as follows:

ITOS - an integer for the top of the operand stack (TOS) is stored in register(s)

LTOS - a long integer for the TOS is stored in register(s)

FTOS - a single-precision floating point for the TOS is stored in register(s)

DTOS - a double-precision floating point for the TOS is stored in register(s)

VTOS - void TOS, meaning the TOS is not currently stored in register(s)

As indicated above, the VTOS state is different from the rest of the states because it indicates that the top of the operand stack is not currently stored in any of the registers. It should be apparent that as data values are pushed onto and popped off of the operand stack, the interpreter may alternate between the VTOS state and one of the other states.

In order to assist in managing the different states of the interpreter, a template table (data structure) 501 shown in Fig. 5 is utilized in some embodiments of the invention. Template table 501 is a table that is indexed by bytecodes 503 and includes fields 505, 507, and 509. Although template table 501 may have over two hundred records (e.g., one for each bytecode), only a subset are shown which are thought to best illustrate the invention.

The virtual machine instructions (or bytecodes) are utilized to index the template table 501. Field 505 stores the state of the interpreter that is expected before virtual machine instructions 503 execute. For example, before an ISTORE bytecode is executed (i.e., store the integer on the top of the operand stack), it is expected that the interpreter will be in the ITOS state indicating that there is an integer for the top of the operand stack stored in the one or more registers. If the interpreter is not in the expected state during execution, that does not necessarily indicate there is an error, but as will described in more detail below, preferred embodiments of the invention are able to detect many errors in the bytecode sequence.

Field 507 stores pointers to functions that generate computer code (or a "template" and hence "template table") for the interpreter to execute virtual machine instructions 503.

In a preferred embodiment, the names of the functions are the same as the name of the bytecodes and the functions are written in the C++ programming language.

Lastly, field 509 stores the current state of the interpreter after execution of virtual machine instructions 503. For example, after an ISTORE bytecode is executed, the current state of the interpreter would be VTOS since the integer stored in the one or more registers has been popped off the operand stack.

In preferred embodiments, field 505 stores the state of the interpreter that is expected before a virtual machine instruction executes and field 509 stores the current state of the interpreter after the virtual machine instruction executes. However, in other embodiments field 505 stores the state of the interpreter that is expected before the template function specified in field 507 executes and field 509 stores the current state of the interpreter after the template function executes. In other words, the state of the interpret may be based on the template functions instead of the virtual machine instructions.

The template table has been described but during interpreter generation, another table (the "dispatch table") is utilized in conjunction with the template table. Fig. 6 shows a layout of a dispatch table 601 which is indexed by virtual machine instructions 603 and includes fields 605, 607, 609, 611, and 613. Field 605 stores pointers to a location or address in the interpreter that executes the indexed virtual machine instructions for the ITOS state. Similarly, fields 607, 609, 611, and 613 store pointers to a location or address in the interpreter that executes the indexed virtual machine instructions for the LTOS, FTOS, DTOS, and VTOS states, respectively. Accordingly, each field is associated with a state of the interpreter. The values of the fields are not shown as they are pointers to within a generated interpreter.

Recalling that the interpreter is typically a software program itself, the dispatch table is a jump table to different locations within the computer code of the interpreter program. In other words, once the next bytecode to be executed is fetched, the interpreter jumps to the location indicated in the dispatch table specified by the next bytecode (utilized as an index) and the current state of the interpreter which specifies one of fields 605, 607, 609, 611, or 613 for the location of the jump. Thus, the program counter of the interpreter is set to the

specified address in the dispatch table. In a preferred embodiment, dispatch table 601 is implemented as five single-dimensional tables, one for each interpreter state.

It should be apparent that the template table and the dispatch table may be implemented as one table (or more than two tables for that matter). However, in preferred embodiments, the template table and dispatch table are separate tables as the template table may be utilized solely during interpreter generation and therefore may be discarded after the interpreter is generated. The dispatch table is generated or filled during interpreter generation and advantageously utilized during interpreter execution. Nevertheless, the information in these tables may be implemented in any number of ways in any number of data structures known to those of skill in the art.

Now that the template and dispatch tables have been described, it may be appropriate to describe how the interpreter may be generated. Fig. 7 shows a process of generating an interpreter. In general, the process generates the interpreter by cycling through all the virtual machine instructions and interpreter state combinations. This may be implemented with nested loops, a single loop or other control structures. In a preferred embodiment, nested loops are utilized.

At step 701, the computer system selects a virtual machine instruction (e.g., by one iteration through a loop through the virtual machine instructions). The system selects an interpreter state at step 703. Once a virtual machine instruction and an interpreter state are selected, the rest of the process in Fig. 7 generates computer code for the interpreter that will handle the selected virtual machine instruction when the interpreter is currently in the selected state. Although the drawings show flowcharts for embodiments of the invention for purposes of illustration, no specific ordering or combination of steps should be implied. In general, steps may be reordered, combined or deleted without departing from the scope of the invention.

At step 705, the system determines if the selected virtual machine instruction and interpreter state are legal. In one embodiment, this is accomplished by determining the expected state of the interpreter for the selected bytecode utilizing the template table (see Fig.

5). If the expected state is the same as the selected state, then the combination of selected virtual machine instruction and state is legal.

If the expected state is different from the selected state, this does not necessarily mean that the combination is illegal. Instead, the system determines if there is a legal way (meaning that does not corrupt the operand stack) from the selected state to the expected state. For example, if the expected state is ITOS and the selected state is VTOS, the interpreter may be put in the ITOS state by moving the top data value in the operand stack that is stored in memory into one or more registers (e.g., store the data value pointed to by SP' into a register and then decrement SP'). As another example, if the expected state is ITOS and the selected state is DTOS, there is no legal way to put the interpreter in the ITOS state since the top of the operand stack currently is a double-precision floating point.

In general, it is legal to go from the state of VTOS to any other state or to go from any other state to VTOS. The reason is that these shifts of interpreter state typically include moving a data value from memory to one or more registers, or vice versa.

If the selected virtual machine instruction and interpreter state are legal, the system may generate prolog computer code at step 707. The prolog computer code is any code that would be advantageously generated before execution of the selected virtual machine instruction. In general, the prolog may depend on the selected virtual machine instruction and the selected interpreter state. For example, if the expected state of the interpreter (for the selected virtual machine instruction) is different than the selected interpreter state, the prolog may include computer code to put the interpreter in the expected state. If the expected and selected states of the interpreter are the same, it may not be necessary to generate prolog computer code.

At step 709, the system calls the template function for the selected virtual machine instruction in order to generate computer code for the interpreter to execute the selected virtual machine instruction. In a preferred embodiment, the template function is called by indexing the template table shown in Fig. 5 with the selected virtual machine instruction. The field, field 507, which stores the pointer to (or address for) the template function is then accessed and the template function is called.

The template function generates computer code to execute the selected virtual machine instruction. It may be helpful to discuss a few examples of template functions. As mentioned earlier, in a preferred embodiment the template functions are written in the C++ and Java programming languages for an x86 microprocessor. The following is a template function for the bytecode ILOAD:

```
void TemplateTable::iload(int n) {
        assembler.movl(eax, address(n));
}
```

The ILOAD method is defined for a class called TEMPLATE_TABLE for the Java virtual machine. As the ILOAD bytecode pushes an integer onto the operand stack, the template function by the same name has a parameter that is an integer. The MOVL method is a C++ function for an ASSEMBLER object that pushes the value of N onto the operand stack by placing it in a register. Recall that MOVL corresponds to the x86 assembly language instruction that moves a 32 bit quantity which is an integer in the Java virtual machine instruction but a long integer in the x86 microprocessor.

As another example, the following is a template function for the bytecode IADD:

```
void TemplateTable::iadd() {
        assembler.popl(edx);
        assembler.addl(eax, edx);
}
```

The IADD method is defined for a class called TEMPLATE_TABLE for the Java virtual machine. The expected state for the IADD bytecode is ITOS so there should be an integer at the top of the operand stack stored in a register (EAX in this example). First, the value pointed to by the SP' pointer (which may be the ESP pointer in the x86 microprocessor) is popped off the stack utilizing the POPL method. It is important to understand that the stack we are discussing now is the native stack stored in memory on the target microprocessor (see left side of Figs. 4B and 4C). Thus, the data value pointed to by the SP' pointer is moved into the register EDX and SP' is then decremented.

At this point, the top of the operand stack is stored in the register EAX and the next highest data value on the operand stack is stored in EDX. The ADDL method corresponds to an assembly language instruction that adds the values stored in EAX and EDX, storing the sum in EAX. The EAX register now stores the desired sum in the appropriate register for the top of the operand stack, meaning the interpreter is now in the ITOS state as specified in the template table of Fig. 5 following the execution of the selected function IADD.

As illustrated above, in preferred embodiments, an object is instantiated for the assembler which includes methods for each assembly language instruction that will be utilized in the interpreter. For simplicity, the names of the methods are the same as the assembly language instructions. It has been found that utilizing an assembler object is beneficial for generation of the interpreter because an extra assembler need not be utilized. In some embodiments, the template functions may be written in assembly language for the desired computer architecture.

At step 711, the system generates epilog computer code. The epilog is computer code that sets up the interpreter to execute the next virtual machine instruction. Thus, the epilog performs the advance and dispatch steps of the interpreter described earlier.

Initially, the prolog computer code fetches the next virtual machine instruction. Since the current state of the interpreter after the execution of the selected virtual machine instruction is known (e.g., from field 509 of the template table in Fig. 5), the next virtual machine instruction may be utilized as an index (or offset) into the dispatch table of Fig. 6 in order to determine the location within the interpreter to execute the next virtual machine instruction. The computer code in the epilog will be discussed in more detail in reference to Fig. 8, but in general, the epilog depends on the selected virtual machine and the current interpreter state.

The system determines if there are more virtual machine instruction/interpreter states for which to generate computer code for the interpreter at step 701. If there are the process returns to step 701 and performs another iteration.

Back at step 705 if it is determined that the selected virtual machine instruction and interpreter state are illegal, the system may generate computer code to handle the error at step 715. Generally speaking, the error is an illegal bytecode sequence. In a preferred embodiment, computer code is generated for the interpreter that jumps to instructions that inform the user that this error has occurred. Although the number of errors detected in this manner will not be as numerous as those detected by a bytecode verifier, it may be desirable especially if one is not required or able to use a bytecode verifier. In some embodiments, the error checking and handling steps 707 and 715 may be omitted.

Fig. 8 shows a process of generating the epilog computer code for the interpreter. The epilog computer code is generated at step 711 of Fig. 7, but a specific embodiment that generates the epilog computer code will be discussed in reference to Fig. 8. At step 801, computer code that fetches the next virtual machine instruction is generated. The next virtual machine instruction may be fetched by incrementing the current bytecode pointer to the next bytecode and then fetching the bytecode pointed to or referenced by the bytecode pointer. As the size of the virtual machine instructions may vary as in the case of Java bytecodes, in a preferred embodiment a table is utilized to store the size of each bytecode so that the bytecode pointer may be incremented by the size in the table to point to the next bytecode.

Once the next virtual machine instruction is fetched, the system generates computer code to calculate an offset into the dispatch table at step 803. The offset is the number which when added to the starting address of the dispatch table of Fig. 6 results in the field indexed by the next bytecode and the current interpreter state. In a preferred embodiment, the dispatch table includes five single-dimensional tables (or subtables), one for each interpreter state. The current state of the interpreter determines which subtable to utilize. The size of each field in the subtables may be a fixed size (e.g., four bytes) so calculating the offset includes multiplying the next bytecode value by the fixed size. In other embodiments where the dispatch table is a single two-dimensional table, numerous techniques well known to those of skill in the art of calculating offsets into two-dimensional arrays may be utilized. Furthermore, the invention is not limited to tables but may be

implemented utilizing any number of data structures including linked lists, hash tables, and the like.

At step 805, the system generates computer code to jump to the location or address in the interpreter specified by the field at the offset in the dispatch table. The dispatch table is a jump table storing addresses within the computer code of the interpreter itself. During interpretation, the epilog computer code performs the advance and dispatch steps for the interpreter. However, other embodiments may place the advance step in the prolog and the dispatch step in the epilog; therefore, the invention is not limited to the specific implementation described herein.

The above has described preferred embodiments of the present invention. Conceptually, one may think that there are five separate interpreters generated: one for each of the interpreter states ITOS, LTOS, FTOS, DTOS, and VTOS. However, in practice, many of the virtual machine instruction/interpreter state combinations are illegal so five separate interpreters are not generated. Furthermore, computer code that executes the virtual machine instructions may be shared so an interpreter according to the present invention may not be much larger in size than a conventional interpreter. In order to more clearly see how computer code that executes the virtual machine instructions may be shared, it may be helpful to the reader to describe how computer code for a sample bytecode may be generated for the interpreter.

Example

As described in reference to Fig. 7, an interpreter may be generated by cycling or looping through the possible virtual machine instruction and interpreter state combinations. As the interpreter is generated, a section of computer code is generated for each virtual machine instruction with a dispatch table being utilized during interpreter execution to hold the entry or jump points for different virtual machine instruction and interpreter state combinations. Therefore, the generated interpreter may include a dispatch table and a sequence of sections of computer code that execute different virtual machine instructions (or handle errors).

With this example, it will be described how computer code that executes the IADD bytecode is generated. Fig. 9 shows a portion of a dispatch table 901 pertaining to the IADD bytecode. The structure of dispatch table 901 is the same as described in reference to Fig. 6. In short, the dispatch table is indexed by virtual machine instructions 903 and includes fields 905, 907, 909, 911, and 913, one field for each state of the interpreter.

The decimal value for the IADD bytecode is 96 as shown in parenthesis. Pointer A_I points to a location or address in the interpreter that executes the IADD bytecode if the interpreter is in the ITOS state. Similarly, pointers A_L, A_F, A_D, and A_V point to a locations or addresses in the interpreter that execute the IADD bytecode when the interpreter is in the LTOS, FTOS, DTOS, and VTOS states, respectively.

The pointers in dispatch table 901 point to addresses within the sequences of sections of computer code generated for the interpreter. Fig. 10 shows sequences of sections of computer code generated for the interpreter. Each section of computer code executes a specific bytecode . Section 1003 includes computer code for execution the IADD bytecode. As shown, section 1003 includes two POPL instructions and an ADDL instruction. These instructions are assembly language (or native machine instructions) for an x86 microprocessor and were generated as follows.

During interpreter generation, the following sections of assembly language instructions may be generated to execute the IADD byte when the interpreter is in the ITOS or VTOS states:

ITOS	VTOS
POPL EDX	POPL EAX
ADDL EAX, EDX	POPL EDX
<DISPATCH/ADVANCE>	ADDL EAX, EDX
	<DISPATCH/ADVANCE>

For simplicity, the computer code that performs the dispatch and advance steps are not explicitly shown. As shown, the only difference between the two sections of computer code is that there is an additional instruction when the interpreter is in the VTOS state. The POPL EAX pops a value off the stack of the native machine and places it in register EAX.

This instruction was generated as prolog to shift the interpreter from the VTOS state into the ITOS state which is the state which is expected for the IADD bytecode (see step 707 of Fig. 7).

The POPL EDX instruction and the ADDL instruction were generated by the template function IADD() accessed in the template table (see previous example and step 709 of Fig. 7). Additionally, the computer code that implements the dispatch and advance steps are the epilog computer code (see step 711 of Fig. 7). As each section of code differs only by an initial assembly language instruction, pointers may be utilized to access a single section of computer code.

Accordingly, section 1003 includes computer code to execute the IADD bytecode when the interpreter is in either the ITOS or VTOS states. As shown in Fig. 10, pointer A_V from dispatch table 901 in Fig. 9 points to the first instruction in section 1003 so that the initial instruction that puts the interpreter from the VTOS state into the ITOS state is executed. Pointer A_I from the dispatch table points to the second instruction in section 1003 since the interpreter is in the ITOS state. As shown, whether the interpreter is in the VTOS or ITOS state, the computer code specified by pointer A_I will direct the interpreter to execute IADD bytecodes.

States LTOS, FTOS and DTOS for the IADD bytecode in dispatch table 901 represent illegal states for the bytecode. Accordingly, pointers A_L, A_F and A_D pointer to a section 1005 of computer code in Fig. 10 that handles the error. The computer code in section 1005 typically indicates to the user that the interpreter has been placed in an illegal state (see also step 715 of Fig. 7). For simplicity, one section of computer code is shown that handles errors; however, more than one section of computer code (or none if error checking is not desired) may be utilized. Additionally, the sections of computer code the execute virtual machine instructions or handle errors may be arranged in any order.

Having discussed an example, it may be beneficial to describe a process of executing a virtual machine instruction with an interpreter according to one embodiment as shown in Fig. 11. The process shown may be utilized to execute virtual machine instructions in an interpreter generated as described herein. However, the process may be

utilized with interpreters that are generated by other methods so the interpreter generation described should not be taken as limiting the implementation of the interpreter.

At step 1101, a computer system puts the interpreter in the expected state, where the expected state is the interpreter state that is expected before a selected virtual machine instruction is executed. In other embodiments, the expected state is the interpreter state that is expected before the computer code in the interpreter that executes the selected virtual machine instruction is run (e.g., the computer code generated by the template function). Step 1101 occurs during interpreter execution and corresponds to the prolog computer code generated at step 707 of Fig. 7 during interpreter generation. If the system is in the expected state, this step may be omitted.

The system executes the selected virtual machine instruction at step 1103. This step occurs during interpreter execution and corresponds to the computer code generated by the template function at step 709 of Fig. 7 during interpreter generation.

Once the selected virtual machine instruction has been executed, the system fetches the next virtual machine instruction at step 1105. Utilizing the next virtual machine instruction, the system calculates an offset into the dispatch table at step 1107. The current state of the interpreter after execution of the selected virtual machine instruction is known. Therefore, the current state along with the next virtual machine instruction may be utilized to calculate an offset into the dispatch table that specifies the location in the interpreter to execute the next virtual machine instruction. In a preferred embodiment where the dispatch table is implemented as multiple single-dimensional subtables, one for each interpreter state, the current state specifies the subtable and the offset is calculated utilizing the next virtual machine instruction (e.g., virtual machine instruction * a fixed size).

At step 1109, the system jumps to the address or location in the interpreter stored in the field at the offset in the dispatch table. The field may include a pointer to a location in the interpreter that executes the next virtual machine instruction. Thus, the jump may cause the system to go back to step 1101 for the next virtual machine instruction.

Steps 1105, 1107 and 1109 occur during interpreter execution and correspond to the computer code generated by the epilog computer code at step 711 of Fig. 7 during

interpreter generation. It should be noted that with the embodiment of the invention shown in Fig. 11, explicit error checking is not required. Instead, if there is an error, the system may jump to computer code to handle the error at step 1109. Accordingly, error checking may be achieved without an impact on performance.

Conclusion

While the above is a complete description of preferred embodiments of the invention, various alternatives, modifications and equivalents may be used. It should be evident that the invention is equally applicable by making appropriate modifications to the embodiments described above. For example, the embodiments described have been in reference to increasing the performance of the Java virtual machine interpreting bytecodes, but the principles of the present invention may be readily applied to other systems and languages. Therefore, the above description should not be taken as limiting the scope of the invention which is defined by the meets and bounds of the impended claims along with their full scope of equivalents.

subtables, one for each interpreter state, the current state specifies the subtable and the offset is calculated utilizing the next virtual machine instruction (e.g., virtual machine instruction * a fixed size).

At step 1109, the system jumps to the address or location in the interpreter stored in the field at the offset in the dispatch table. The field may include a pointer to a location in the interpreter that executes the next virtual machine instruction. Thus, the jump may cause the system to go back to step 1101 for the next virtual machine instruction.

Steps 1105, 1107 and 1109 occur during interpreter execution and correspond to the computer code generated by the epilog computer code at step 711 of Fig. 7 during interpreter generation. It should be noted that with the embodiment of the invention shown in Fig. 11, explicit error checking is not required. Instead, if there is an error, the system may jump to computer code to handle the error at step 1109. Accordingly, error checking may be achieved without an impact on performance.

Conclusion

While the above is a complete description of preferred embodiments of the invention, various alternatives, modifications and equivalents may be used. It should be evident that the invention is equally applicable by making appropriate modifications to the embodiments described above. For example, the embodiments described have been in reference to increasing the performance of the Java virtual machine interpreting bytecodes, but the principles of the present invention may be readily applied to other systems and languages. Therefore, the above description should not be taken as limiting the scope of the invention which is defined by the meets and bounds of the impended claims along with their full scope of equivalents.

4. Brief Description of Drawing

Fig. 1 illustrates an example of a computer system that may be utilized to execute the software of an embodiment of the invention.

Fig. 2 shows a system block diagram of the computer system of Fig. 1.

Fig. 3 shows how a Java source code program is executed.

Fig. 4A shows a stack; Fig. 4B shows an operand stack of the present invention where the value for the top of the operand stack is stored in a register; and Fig. 4C shows an operand stack of the present invention where multiple registers and registers for storing different data types may store the value for the top of the operand stack.

Fig. 5 illustrates a template table utilized during interpreter generation to organize interpreter states and template functions.

Fig. 6 illustrates a dispatch table generated during interpreter generation that stores pointers to locations within the interpreter utilized to direct interpreter execution flow.

Fig. 7 shows a process of generating an interpreter that utilizes a state of the interpreter to indicate that data type of the value for the top of the operand stack that is store in one or more registers.

Fig. 8 shows a process of generating epilog computer code that executes the advance and dispatch steps of the interpreter.

Fig. 9 shows a portion of a dispatch table of Fig. 6 for executing the bytecode IADD.

Fig. 10 shows sections of computer code for interpreter that execute virtual machine instructions and handle errors.

Fig. 11 shows a process of executing a virtual machine instruction with an interpreter according to an embodiment of the present invention.

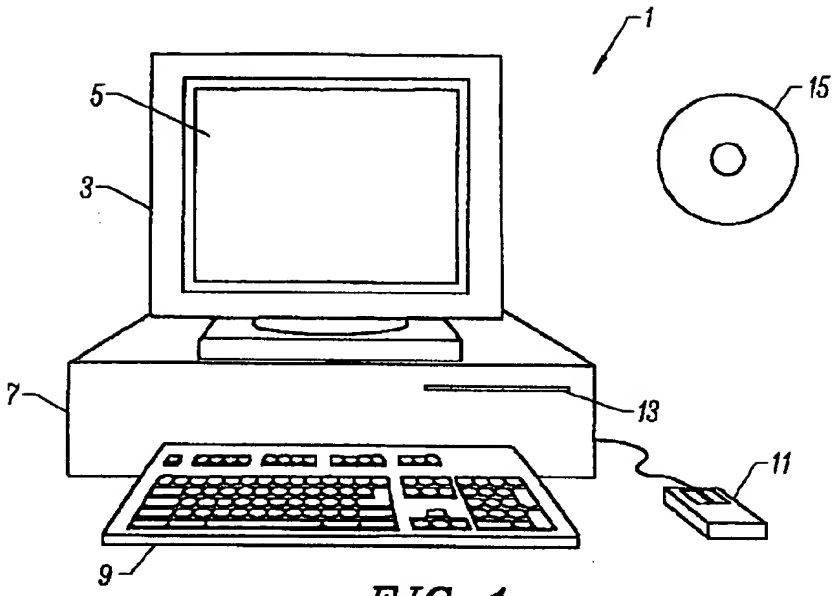


FIG. 1

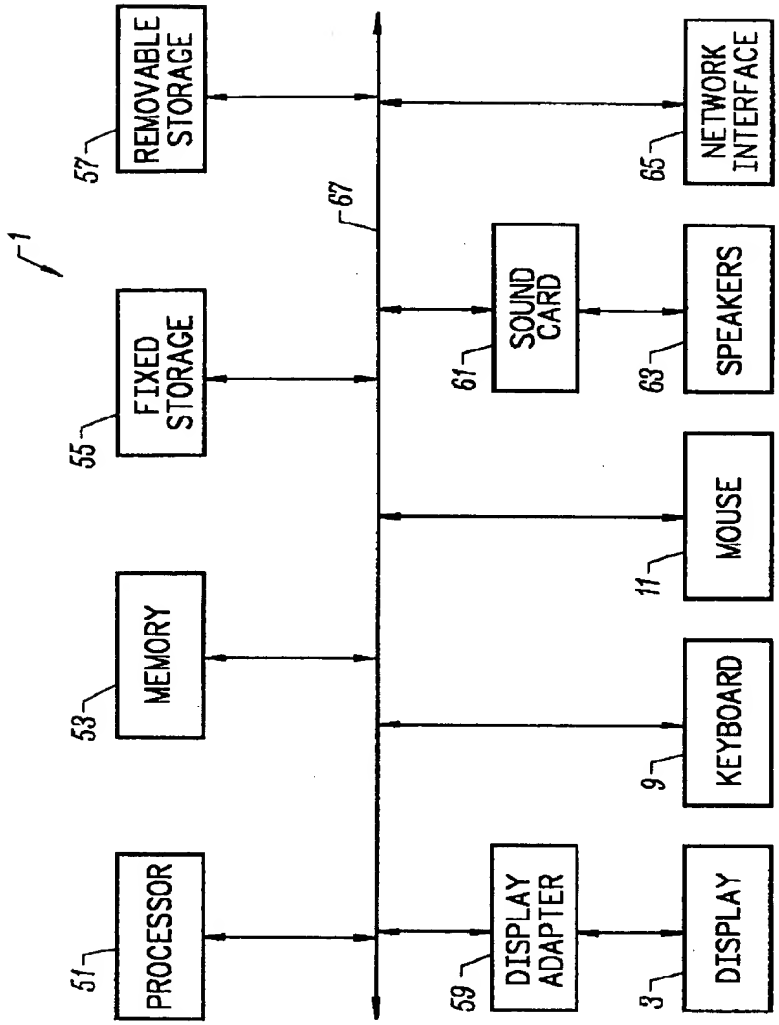

1
67
REMOVABLE STORAGE
57
FIXED STORAGE
55
MEMORY
53
PROCESSOR
51
SOUND CARD
61
DISPLAY ADAPTER
59
NETWORK INTERFACE
65
SPEAKERS
63
MOUSE
11
KEYBOARD
9
DISPLAY
3

FIG. 2

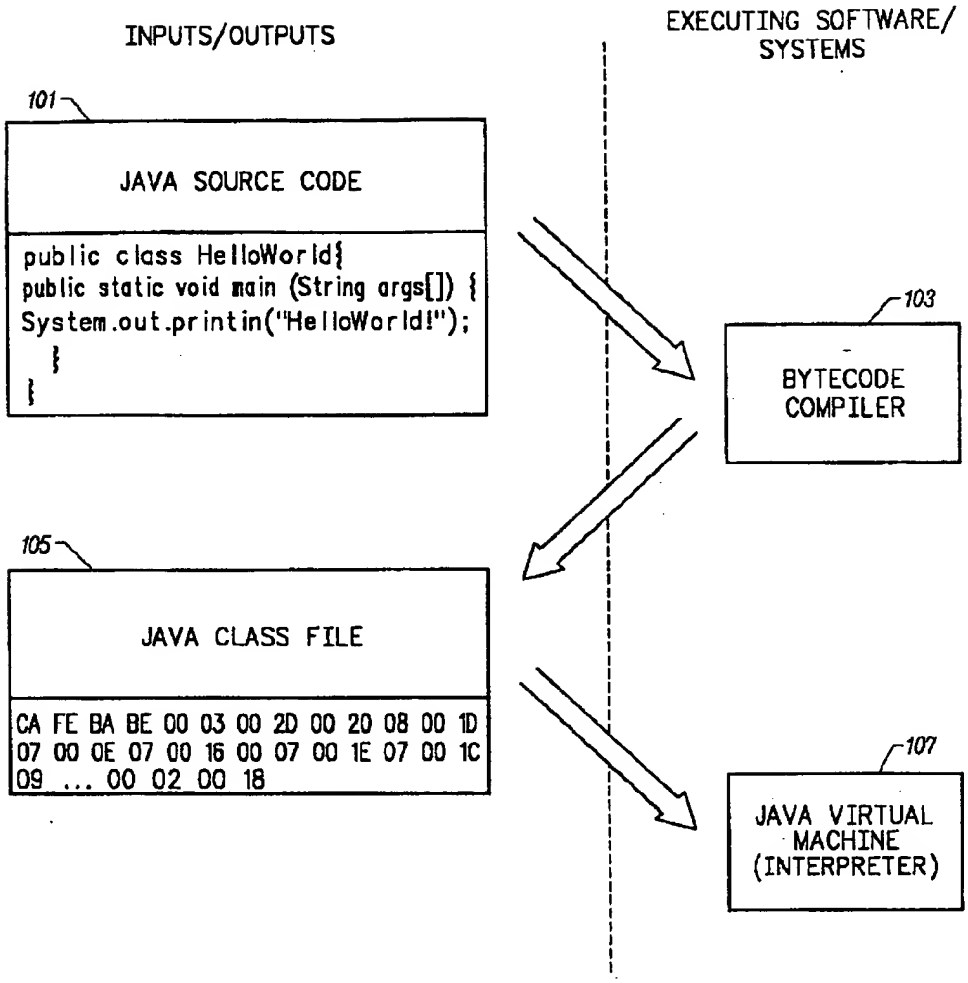

INPUTS/OUTPUTS
EXECUTING SOFTWARE/ SYSTEMS
101
JAVA SOURCE CODE
public class HelloWorld{
public static void main (String args[]) {
System.out.println("HelloWorld!");
}
}
103
BYTECODE COMPILER
105
JAVA CLASS FILE
CA FE BA BE 00 03 00 2D 00 20 08 00 1D
07 00 0E 07 00 16 00 07 00 1E 07 00 1C
09 ... 00 02 00 18
107
JAVA VIRTUAL MACHINE (INTERPRETER)

FIG. 3

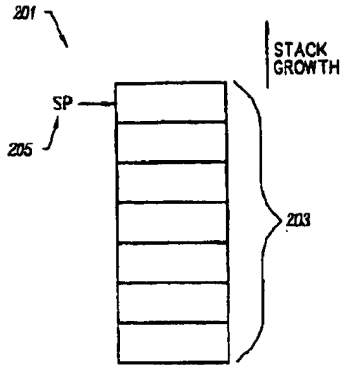


FIG. 4A

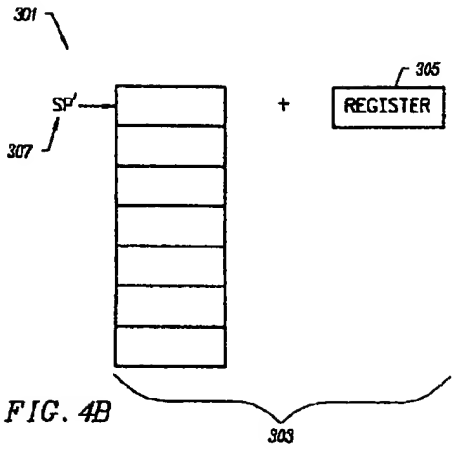


FIG. 4B

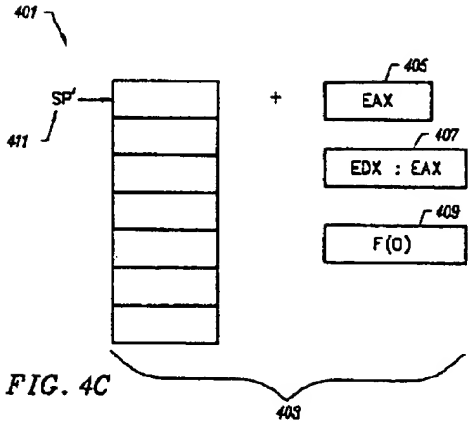


FIG. 4C

TEMPLATE TABLE

501 BYTECODE	STATE BEFORE	TEMPLATE FUNCTION	STATE AFTER
0	<ANY>	NOP( )	<SAME>
...	...		
21	VTOS	ILOAD( )	ITOS
22	VTOS	LLOAD( )	LTOS
23	VTOS	FLOAD( )	FTOS
24	VTOS	DLOAD( )	DTOS
...	...		
54	ITOS	ISTORE( )	VTOS
55	LTOS	LSTORE( )	VTOS
56	FTOS	FSTORE( )	VTOS
57	DTOS	DSTORE( )	VTOS
...			
96	ITOS	IADD ( )	ITOS
97	LTOS	LADD ( )	LTOS
98	FTOS	FADD ( )	FTOS
99	DTOS	DADD ( )	DTOS
...	...		

503 505 507 509

FIG. 5

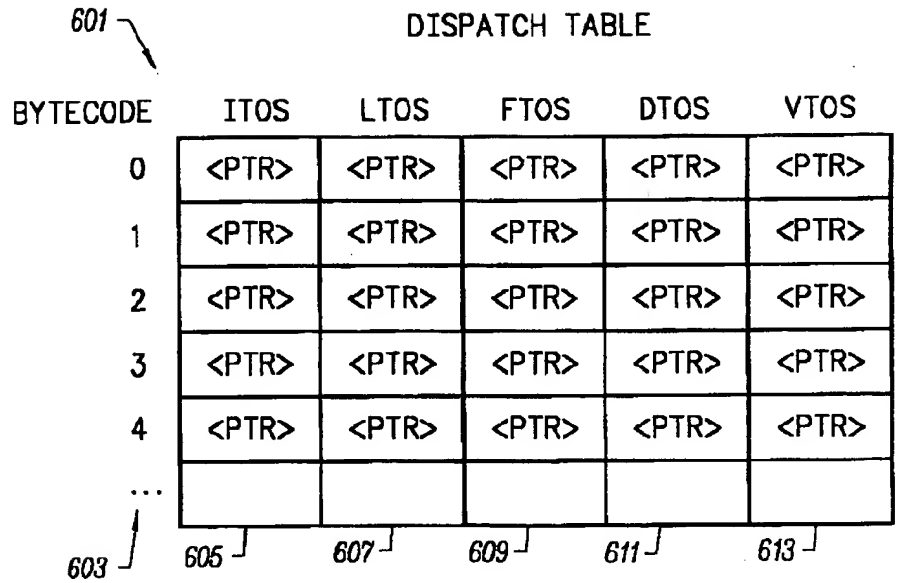

601
DISPATCH TABLE
BYTECODE
ITOS
LTOS
FTOS
DTOS
VTOS
0
1
2
3
4
...
<PTR>
603
605
607
609
611
613

FIG. 6

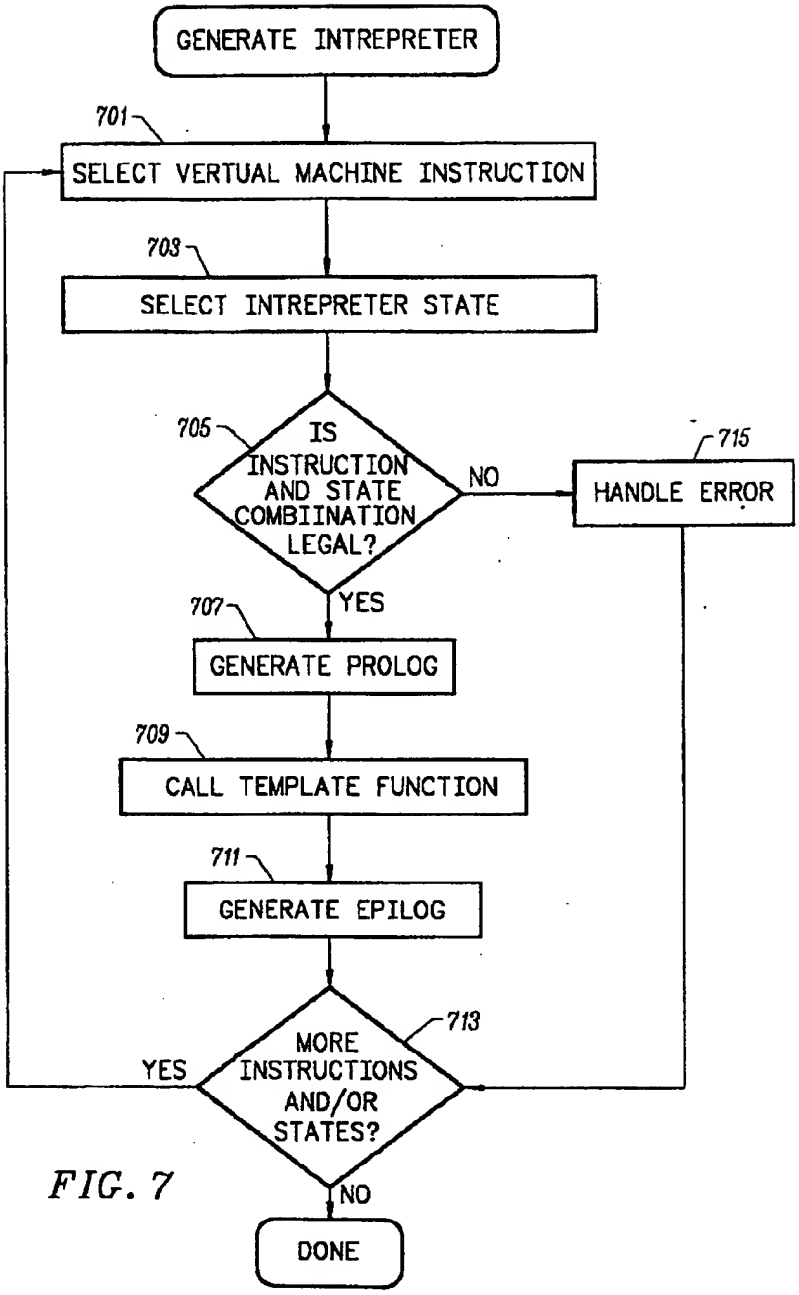

GENERATE INTREPRETER
701
SELECT VERTUAL MACHINE INSTRUCTION
703
SELECT INTREPRETER STATE
705
IS INSTRUCTION AND STATE COMBIINATION LEGAL?
NO
715
HANDLE ERROR
YES
707
GENERATE PROLOG
709
CALL TEMPLATE FUNCTION
711
GENERATE EPILOG
713
MORE INSTRUCTIONS AND/OR STATES?
YES
NO
DONE

FIG. 7

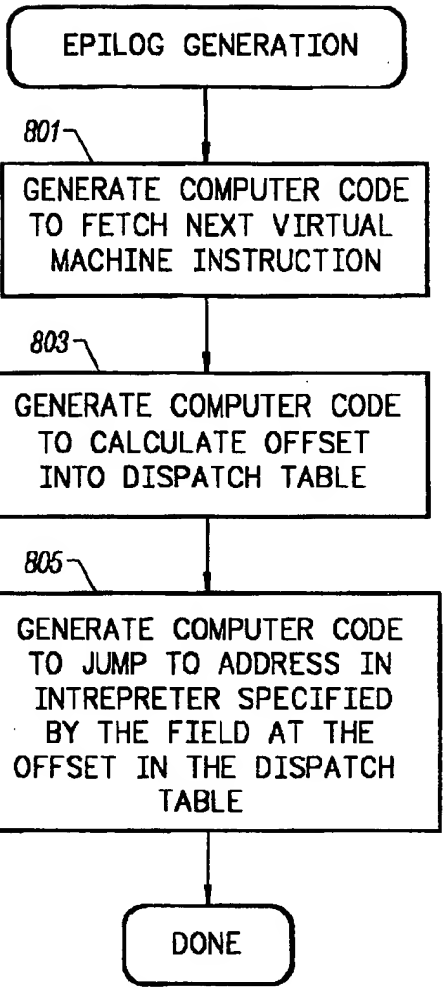

EPILOG GENERATION
801
GENERATE COMPUTER CODE TO FETCH NEXT VIRTUAL MACHINE INSTRUCTION
803
GENERATE COMPUTER CODE TO CALCULATE OFFSET INTO DISPATCH TABLE
805
GENERATE COMPUTER CODE TO JUMP TO ADDRESS IN INTREPRETER SPECIFIED BY THE FIELD AT THE OFFSET IN THE DISPATCH TABLE
DONE

FIG. 8

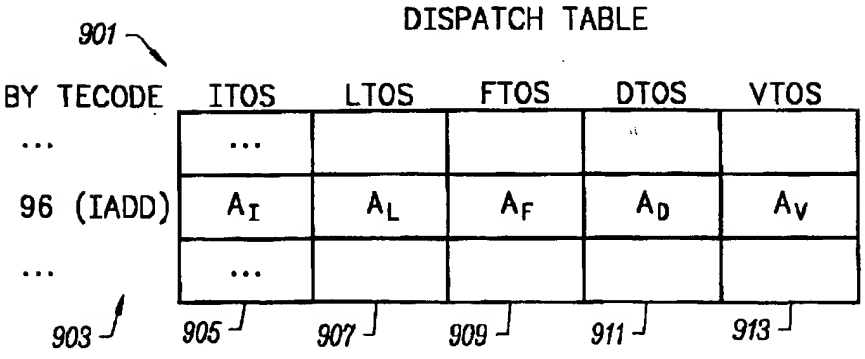

DISPATCH TABLE
901
BY TECODE
ITOS
LTOS
FTOS
DTOS
VTOS
...
96 (IADD)
A_I
A_L
A_F
A_D
A_V
903
905
907
909
911
913

FIG. 9

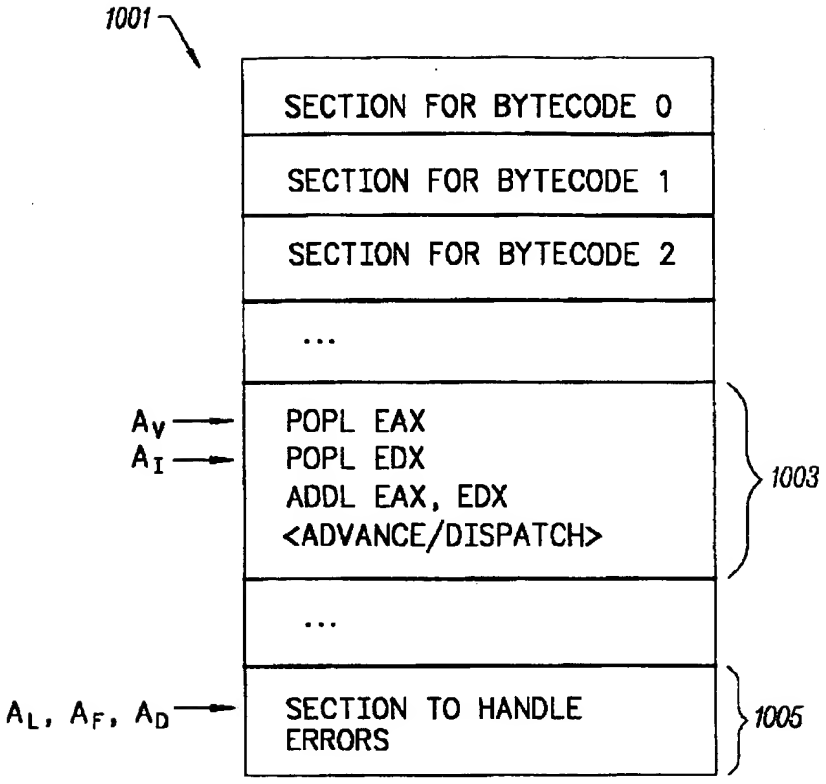

1001
SECTION FOR BYTECODE 0
SECTION FOR BYTECODE 1
SECTION FOR BYTECODE 2
...
A_V
A_I
POPL EAX
POPL EDX
ADDL EAX, EDX
<ADVANCE/DISPATCH>
1003
...
A_L, A_F, A_D
SECTION TO HANDLE ERRORS
1005

FIG. 10

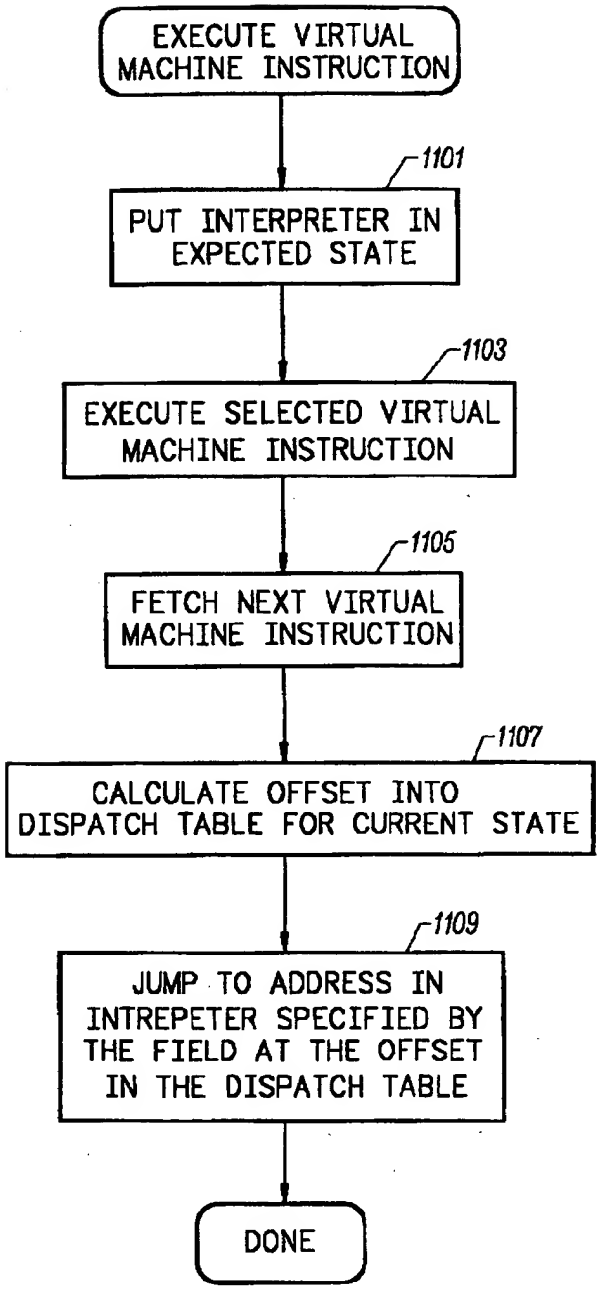


FIG. 11

1. Abstract

Systems and methods for increasing the execution speed of interpreted programs which utilize an operand stack are provided. The value for the top of the operand stack is stored in one or more registers. A state of the interpreter indicates the data type of the value for the top of the operand stack stored in the one or more registers. An interpreter may be generated that is both fast and efficient in terms of the memory required for the interpreter.

2. Representative Drawing

Fig. 4